信息科学与技术丛书

石庆冬　编著

实用Linux Shell编程

- 强调动手动脑，实践与理论并重
- 不同难度的实例，适合各层次的读者
- 原创内容与程序员的开发工作紧密相关
- IT 界一线人士的真知灼见

机械工业出版社
CHINA MACHINE PRESS



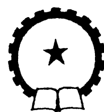
VISIT…

LANZAROTE
Caliente.COM

信息科学与技术丛书

实用 Linux Shell 编程

石庆冬　编著



机 械 工 业 出 版 社

本书系统地介绍了在 Linux 系统中广泛使用的 Bash 脚本语言的基本知识。全书内容的安排由浅入深、体系合理。先讲解脚本的概念和学习环境的搭建，接下来介绍 Linux 的常用命令，然后根据概念之间的依赖关系，讲解 Bash 环境设置、变量与数组、条件流程控制、循环、函数、正则表达式、文本处理、进程与作业、高级话题等。

本书适合 Linux 脚本编程零基础的读者学习。可以作为高校、职校计算机专业的参考教材，也可以作为从事 Linux 软件相关工作的工程师的参考读物或培训教材，同时还是广大 Linux 脚本爱好者不可多得的一本中文原创读物。

书中代码可在 http://www.cmpbook.com 免费下载。

图书在版编目（CIP）数据

实用 Linux Shell 编程 / 石庆冬编著．—北京：机械工业出版社，2014.11
（信息科学与技术丛书）
ISBN 978-7-111-48202-4

Ⅰ．①实…　Ⅱ．①石…　Ⅲ．①Linux 操作系统-程序设计　Ⅳ．①TP316.89

中国版本图书馆 CIP 数据核字（2014）第 230876 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）
责任编辑：车　忱
责任校对：张艳霞
责任印制：李　洋

北京振兴源印务有限公司印刷

2014 年 11 月第 1 版・第 1 次印刷
184mm×260mm・19.25 印张・476 千字
0001－3000 册
标准书号：ISBN 978-7-111-48202-4
定价：52.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务	网络服务
社服务中心：（010）88361066	教 材 网：http://www.cmpedu.com
销 售 一 部：（010）68326294	机工官网：http://www.cmpbook.com
销 售 二 部：（010）88379649	机工官博：http://weibo.com/cmp1952
读者购书热线：（010）88379203	**封面无防伪标均为盗版**

出 版 说 明

随着信息科学与技术的迅速发展，人类每时每刻都会面对层出不穷的新技术和新概念。毫无疑问，在节奏越来越快的工作和生活中，人们需要通过阅读和学习大量信息丰富、具备实践指导意义的图书来获取新知识和新技能，从而不断提高自身素质，紧跟信息化时代发展的步伐。

众所周知，在计算机硬件方面，高性价比的解决方案和新型技术的应用一直备受青睐；在软件技术方面，随着计算机软件的规模和复杂性与日俱增，软件技术不断地受到挑战，人们一直在为寻求更先进的软件技术而奋斗不止。目前，计算机和互联网在社会生活中日益普及，掌握计算机网络技术和理论已成为大众的文化需求。由于信息科学与技术在电工、电子、通信、工业控制、智能建筑、工业产品设计与制造等专业领域中已经得到充分、广泛的应用，所以这些专业领域中的研究人员和工程技术人员越来越迫切需要汲取自身领域信息化所带来的新理念和新方法。

针对人们了解和掌握新知识、新技能的热切期待，以及由此促成的人们对语言简洁、内容充实、融合实践经验的图书迫切需要的现状，机械工业出版社适时推出了“信息科学与技术丛书”。这套丛书涉及计算机软件、硬件、网络和工程应用等内容，注重理论与实践的结合，内容实用、层次分明、语言流畅，是信息科学与技术领域专业人员不可或缺的参考书。

目前，信息科学与技术的发展可谓一日千里，机械工业出版社欢迎从事信息技术方面工作的科研人员、工程技术人员积极参与我们的工作，为推进我国的信息化建设做出贡献。

机械工业出版社

前　言

计算机操作系统主要分为 Windows 和 UNIX/Linux 两大阵营，IT 技术人员对这两种操作系统都应该熟悉。学习免费的、开源的、能在 PC 上运行的 Linux 操作系统的必要性无需赘言。学习 Linux 都是从执行键盘字符命令的地方——shell（外壳）开始。shell 包裹着 Linux 的内核和硬件，并提供了一套用于和内核通信的指令集，shell 本身也是一门脚本程序设计语言。Linux 下包含多种 shell，而 Bash 是 Linux 的默认 shell，所以掌握 Bash 编程是深入学习 Linux 的前提条件。

本书深入浅出、系统地介绍了在 Linux 系统中进行 Bash 编程的基本知识。

第 1 章介绍脚本的概念、学习环境的搭建、Linux 命令的格式和获得命令帮助的途径。

第 2 章介绍常用的 Linux 命令和基础知识，包括软链接和硬链接、权限控制、SUID 与 SGID、文件的三种时间和 vi 的使用等。

第 3 章介绍 Bash 环境的设置和几个常用的内置命令，还介绍了管道、输入输出重定向和文件描述符等概念。本章还讲解了如何写出一个可执行的脚本。

第 4 章介绍变量与数组。包括变量的定义、变量的读取、变量的导出、只读变量、整型变量的运算、索引数组、关联数组、环境变量、内置变量和字符串的操作等。

第 5 章讲解条件流程控制。包括整型数关系运算、字符串关系运算、文件属性条件判断、逻辑的与或非、命令的与或非、if 与 case——两种控制程序流程的命令结构，最后介绍了 null 命令的应用。

第 6 章讲解循环。包括 for、while 和 until 循环，控制循环的命令 break、continue 和 shift。还讲解了产生字符菜单选项的 select 命令和用来处理脚本选项和参数的 getopts 命令。

第 7 章讲解函数。包括函数的定义、函数参数的处理、函数内的局部变量、函数的导出、函数的返回和递归函数等。

第 8 章介绍正则表达式的概念和文本处理的常用方法。包括过滤命令 grep 和 egrep、剪切内容命令 cut、合并行内容命令 paste、转换或删除字符命令 tr、排序命令 sort、流编辑器 sed、文本扫描和处理语言 awk 和格式化打印命令 printf。

第 9 章讲解进程与作业。包括进程与作业的查看、挂起作业、将作业放置到前台或后台、给作业发送信号的命令 kill、捕获信号的命令 trap、等待命令 wait、移除作业命令 disown 和暂停 shell 的命令 suspend 等。

第 10 章介绍一些 Bash 中的高级应用。包括二次扫描命令 eval、目录栈命令、并行命令 parallel、（非）登录与（非）交互 shell 的概念和 shell 的选项（包括 Bash 自带的选项和可通过 set 与 shopt 命令设置的选项）。最后，介绍了脚本调试的基本方法和几个脚本实例。

附录简单介绍了 Bash 的 shellshock 漏洞。

本书强调实践。每讲解一个命令或概念时，几乎都是通过例子引出或者用实例解释，而不是将其官网手册上的选项和参数列表的中文翻译全盘给出。本书基于 Ubuntu 操作系统，所有的例子都在 Ubuntu 下经过了验证。

本书作者在大型企业从事软件工作多年。在学习和使用 Bash 的过程中，作者发现：有关 Bash 的资料、译著较多，网络上志愿者们的心得体会或转载的资料很多，但中文的、原创的图书比较少。本书对于作者而言是多年学习和使用 Bash 的一个总结，现在整理成册并出版，一方面可以帮助他人学习，为大家奉献一本 Bash 中文原创读物；另一方面是对自己学习效果的检查和进一步学习 Bash 的鞭策。

原中国煤炭报总编室主任谢国实教授、诺基亚资深工程师邹德禹、邱海泓，对本书进行了非常认真的审阅，并提出了很多中肯的修改意见，在此深表感谢！

希望广大读者朋友在阅读过程中，如果发现缺陷和不妥之处，能提出宝贵的建议和意见。您的反馈，很可能就是此书将来再版的依据和动力！

作　者

2014 年春于北京

目　　录

出版说明
前言
第 1 章　概述 ······ 1
1.1　为什么要学习 Linux shell 脚本 ······ 1
1.2　什么是 shell ······ 1
1.3　什么是 shell 脚本 ······ 2
1.4　为什么要学习 Bash ······ 3
1.5　Bash 学习环境的准备 ······ 4
1.5.1　Linux 的准备 ······ 4
1.5.2　Cygwin 工具 ······ 4
1.5.3　VMware 与 Linux 虚拟机 ······ 7
1.6　Linux 命令格式简介 ······ 10
1.7　如何获得 Linux 命令的帮助 ······ 11
第 2 章　Linux 基础知识与常用命令 ······ 15
2.1　Linux 的文件与目录结构 ······ 15
2.2　查看文件清单命令 ls ······ 16
2.3　浏览文件命令 cat、more、less、head 和 tail ······ 18
2.4　文件统计命令 wc ······ 19
2.5　改变当前工作目录命令 cd ······ 20
2.6　创建目录命令 mkdir ······ 23
2.7　复制命令 cp ······ 24
2.8　重命名或移动命令 mv ······ 25
2.9　创建符号链接和硬链接命令 ln ······ 26
2.10　显示当前目录命令 pwd ······ 30
2.11　产生空文件或者改变文件时间戳命令 touch ······ 30
2.12　查看账户名及其所属组的命令 whoami、id 和 groups ······ 32
2.13　文件与目录的权限 ······ 32
2.14　改变权限命令 chmod ······ 35
2.15　SUID 与 SGID 以及粘滞位 ······ 37
2.15.1　SUID ······ 37
2.15.2　SGID ······ 39
2.15.3　粘滞位 ······ 41
2.16　查看文件的三种时间 ······ 43
2.17　删除命令 rm 与 rmdir ······ 44
2.18　编辑文件命令 vi ······ 45
2.19　用于显示的命令 echo ······ 48

2.20 查看文件类型命令 file …… 49
2.21 显示树状目录信息命令 tree …… 50
2.22 查找命令 find …… 51
第 3 章 Bash 内置命令与环境简介 …… 53
3.1 内置命令与外部命令 …… 53
3.2 认识 Bash 环境 …… 58
3.2.1 命令行提示 PS1 …… 59
3.2.2 搜索路径 PATH …… 60
3.2.3 设置和取消别名命令 alias 和 unalias …… 61
3.2.4 修改.bashrc——设置自己的环境 …… 62
3.3 权限掩码命令 umask …… 64
3.4 source 命令和点命令 …… 66
3.5 编写并运行第一个脚本 …… 67
3.6 命令解释顺序与命令类型的查询 …… 68
3.6.1 命令的解释顺序 …… 68
3.6.2 改变命令解释顺序的三条内置命令 …… 69
3.6.3 命令类型的查询 …… 70
3.7 命令的退出状态以及命令 true 和 false …… 71
3.8 管道 …… 72
3.9 输入输出重定向 …… 73
3.9.1 标准输入和标准输出 …… 73
3.9.2 输入输出重定向与输出追加重定向 …… 74
3.9.3 预防输出重定向覆盖旧文件 …… 77
3.9.4 标准错误输出与“黑洞” …… 78
3.9.5 同时把结果输出至标准输出与文件的命令 tee …… 80
3.10 exec 命令与文件描述符 …… 82
3.11 有关命令的进一步讨论 …… 87
3.11.1 一行多命令 …… 87
3.11.2 将命令放在后台执行 …… 87
3.11.3 命令行的续行 …… 88
3.11.4 命令的补齐与命令历史 …… 89
3.11.5 操作名字含空格的文件 …… 90
3.11.6 操作名字首字母为减号的文件 …… 90
3.11.7 Windows 与 Linux 文件格式的互转 …… 91
3.11.8 小括号与大括号中的命令 …… 92
3.11.9 子 shell …… 92
第 4 章 变量与数组 …… 94
4.1 变量的定义与清除 …… 94
4.2 字符串定义及单双引号与大括号的使用 …… 95

4.3 将命令执行结果赋给变量——反引号与$() …… 98
4.4 键盘输入变量值——内置命令 read …… 100
4.5 整型变量运算 …… 102
4.6 浮点运算 …… 106
4.7 定义只读变量命令 readonly 和 declare -r …… 107
4.8 定义整型变量命令 declare -i …… 107
4.9 数组 …… 109
4.10 关联数组 …… 113
4.11 导出变量命令 export 和 declare -x …… 114
4.12 declare 命令总结 …… 116
4.13 环境变量与特殊变量 …… 116
4.14 内置变量 …… 120
4.15 计算表达式值的命令 expr …… 125
4.16 变量测试及其相应的赋值 …… 127
4.17 字符串操作 …… 132
4.17.1 Bash 内置的字符串操作 …… 132
4.17.2 用命令 expr 处理字符串 …… 135
第 5 章 条件流程控制 …… 138
5.1 条件判断与 test 命令 …… 138
5.1.1 整型数关系运算 …… 138
5.1.2 字符串关系运算 …… 139
5.1.3 文件属性条件判断 …… 141
5.1.4 逻辑的与或非 …… 143
5.1.5 与或非的优先级 …… 145
5.1.6 双中括号格式 …… 145
5.1.7 在双小括号里面进行整数比较 …… 147
5.1.8 命令的与或非 …… 148
5.1.9 判断变量是否定义 …… 150
5.2 条件测试结构 if …… 151
5.3 if-else 结构 …… 153
5.4 if-elif 结构 …… 156
5.5 分情况选择处理——case 命令 …… 159
5.6 命令 exit 与 if 及 case 命令的配合 …… 162
5.7 用 here 文档与 case 命令生成菜单 …… 163
5.8 null 命令 …… 165
第 6 章 循环 …… 169
6.1 for 循环 …… 169
6.2 算术 for 循环 …… 175
6.3 while 循环 …… 177

6.4 until 循环 179
6.5 用 break 和 continue 控制循环 181
6.6 用命令 shift 控制循环 185
6.7 选择命令 select 187
6.8 循环命令与 I/O 重定向及管道的配合 192
6.9 脚本的选项与参数 195
6.9.1 命令 shift 195
6.9.2 命令 getopts 197
第 7 章 函数 202
7.1 函数定义 202
7.2 给函数传递参数 203
7.3 函数的局部与全局变量 204
7.4 当前的函数名 FUNCNAME 206
7.5 在命令行执行函数 207
7.6 查看当前 shell 的函数定义 208
7.7 选项-f 与函数的导出、清除与只读设置 209
7.8 返回命令 return 210
7.9 递归函数 213
第 8 章 正则表达式与文本处理 215
8.1 文件名替换 215
8.1.1 多字符替换* 215
8.1.2 单字符替换? 216
8.1.3 范围替换[]与[!] 217
8.2 正则表达式与 grep 218
8.2.1 过滤器 grep 218
8.2.2 扩展的 egrep 224
8.2.3 POSIX 字符类 224
8.2.4 Bash 扩展模式匹配 226
8.3 剪取内容命令 cut 227
8.4 合并相应行的命令 paste 229
8.5 转换或删除字符命令 tr 231
8.6 排序命令 sort 233
8.7 流编辑器 sed 236
8.7.1 替换命令 s 236
8.7.2 其他命令 240
8.7.3 一行多条命令与保存匹配& 242
8.7.4 sed 的退出状态 243
8.7.5 sed 脚本 244
8.8 文本处理工具 awk 245

8.9 格式化打印命令 printf……251
第 9 章 进程与作业……254
9.1 查看进程命令 ps……254
9.2 挂起进程〈Ctrl+Z〉键……255
9.3 前台 fg 与后台 bg……256
9.4 发送信号命令 kill……257
9.5 等待命令 wait……259
9.6 捕获信号命令 trap……261
9.7 移除作业的命令 disown……264
9.8 暂停 shell 的命令 suspend……265
第 10 章 高级话题……267
10.1 二次扫描命令 eval……267
10.2 目录栈操作命令 pushd、popd 与 dirs……269
10.3 波浪号扩展……272
10.4 （非）登录及（非）交互 shell……273
10.5 Bash shell 选项……273
10.6 用命令 set 设置 shell……274
10.7 用命令 shopt 设置 shell……278
10.8 终端行设置命令 stty……280
10.9 不在脚本和函数内使用别名……282
10.10 Bash 调试……284
10.11 并行命令 parallel……287
10.12 模拟旋转型进度指示……289
10.13 删除文件空行……291
10.14 完善 while-shift 循环……292
附录 Bash 安全漏洞 shellshock……294
参考文献……297

第 1 章　概　　述

Linux 在近些年发展和普及非常迅速，它是一个可以安装在个人电脑上的类 UNIX 操作系统。Linux 的魅力在于它的免费、开源、与互联网的结合，还有在 Linux 下众多的应用程序。Linux 的用户越来越多，甚至挤占了 UNIX 的生存空间，这已经是不争的事实。可以说 Linux 无处不在。Linux 基金会执行委员 Jim Zemlin 说过，“我们每天都在用着 Linux，即使大家不知道。Linux 显然已成为我们生活中不可或缺的一部分了。”

可能有人说他只是用安装了 Windows 系统的个人电脑上网，并未接触到 Linux。他确实没有直接接触 Linux，但他在利用互联网查资料或者与他人沟通时，很可能有他看不到的数台 Linux 服务器在为他工作着。举个大家看得到的例子吧，近几年 Android（安卓）平台手机很流行。Android 就是一种以 Linux 为基础的开源操作系统，主要用于便携设备。假设让全世界使用 Linux 或者与 Linux 相关的设备停止工作一分钟，那么将会有手机无法接打、交通信号灯不工作、股市交易停止、网站瘫痪、核潜艇不听指挥、工厂停工等现象发生。

1.1　为什么要学习 Linux shell 脚本

Linux 是许多工程技术和科研人员必须掌握的基础知识。当前的 Linux 发展得像 Windows 一样，似乎用鼠标和图形界面就可以完成许多工作。

有人说，“我使用过某公司开发的 Linux 下的某图形界面工具，用起来简单，功能强大，单击几个按钮，该工具就可做许多事情。”的确，按一按按钮即可。但是，每单击一下按钮，都可能触发了某个脚本的运行，或者是一个以上的，甚至是一种以上的脚本的执行。因为图形界面工具的开发者完成了许多脚本编码工作，才使得用户得到了简单易用的软件产品。

对于想比较深入地了解 Linux 的人而言，如果只会执行几个常用的 Linux 命令，或者只能在图形 Linux 系统中单击鼠标是不够的。试想，用鼠标加上图形界面如何实现条件判断功能？如何实现嵌套的循环？学习 shell 脚本对于深入地学习 Linux 是很有帮助的，是必须的。一句话，shell 脚本是想深入学习或全面了解 Linux 的人的必修课。

1.2　什么是 shell

shell 是 Linux 的外壳，是 Linux 的用户界面，可以为用户提供输入命令和参数，并可以得到命令执行结果的环境。用户、shell 和操作系统之间的关系如图 1-1 所示。

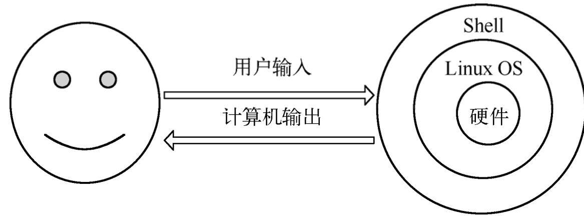


图 1-1　Shell 是包围在 Linux 系统之外的一层外壳

例如，用户想知道现在的时间，打开一个 shell 命令行窗口，也叫 shell terminal（本章后面将讲述 shell 环境的准备），输入命令 date 即可。

```
$ date
Sun Jul 29 11:06:30 EDT 2012
```

看来 date 命令提供的信息不少，时区也显示出来了，EDT 表示美国东部夏令时间（Eastern Daylight Time）。如果只想知道日期或者时间，不希望有其他信息输出，需要输入 date +%F 或者 date +%T。

```
$ date +%F
2012-07-29
$ date +%T
11:12:54
```

再举两个例子。用户想查看当月的日历（calendar），输入命令 cal 即可。

```
$ cal
            July 2012
Su  Mo   Tu   We   Th   Fr   Sa
 1   2    3    4    5    6    7
 8   9   10   11   12   13   14
15  16   17   18   19   20   21
22  23   24   25   26   27   28
29  30   31
```

如果用户想计算 4+5 并打印其结果，可运行 echo $((4+5))命令。

```
$ echo $((4+5))
9
```

根据以上几个例子可以看出，shell 就是输入 Linux 指令并得到计算机输出的地方。Linux kernel 是统一的，但是 shell 有很多种，比较常用的有 Bash（Bourne Again shell）、Csh(C shell)、Tcsh(TENEX C shell)、Ksh(Korn shell)、Zsh(Z shell)等。掌握了一种 Linux shell 之后，学习其他 shell 非常容易。

1.3　什么是 shell 脚本

可执行程序一般有两种实现方式：一种是二进制方式；另一种是脚本（script）方式。

二进制方式是先将编写好的程序（如 C/C++程序）进行编译，变成计算机可识别的指令代码（如.exe 文件），然后再执行。这种编译好的程序只能执行、使用，却看不到它的程序内容。

脚本，简单地说就是由一条一条的命令组成的文本文件，可以用记事本查看或者编辑。脚本程序在执行时，是由系统的一个解释器将其一条条地翻译成计算机可识别的指令，并按程序的逻辑顺序执行（一般来说，脚本程序是确定的一系列控制计算机进行运算操作的动作的组合，在其中可以实现一定的逻辑分支等）。因为脚本在执行时多了一道翻译的过程，所以它比二进制程序执行效率要稍低一些。

Linux shell 脚本，类似于 DOS 批处理（batch）脚本，就是将一些可执行的 Linux 命令放在一个文件里面，然后按照顺序（确切地说，是逻辑顺序，因为会有判断、分支、循环等），命令一条条地执行。例如，接着前面的例子，假设需要同时知道现在的日期、时间和本月的日历，那么需要先后输入相应的 3 条命令，这并不麻烦。但是假设需要知道很多信息，需要输入 30 条命令或者更多，并且每天都要执行同样的命令，是不是就有些麻烦了？如果把这些命令写在一个文本文件里面就不麻烦，该文本文件就是一个脚本。每天只需执行该脚本，写在脚本里面的命令会依次运行。还是接着前面的例子，这 3 条命令组成的脚本如下：

```
#!/bin/bash
date +%F
date +%T
cal
```

该脚本一共 4 行，第一行指明了脚本解释器的位置，接下来的 3 行依次是需要执行的命令。运行该脚本就会得到想要的结果。运行一个脚本，输入脚本文件名即可，仅需要用键盘输入一条命令；如果没有脚本，将需要输入 3 条、30 条或者更多的命令。

但 shell 脚本不能被简单地理解为：将某些 Linux 命令“堆放”在一起。实际上，每种 shell 脚本都是一门程序设计语言。一个包含多条命令的脚本，打开它，从表面上看，命令被“堆放”在一起，但是命令的先后顺序不是任意的，一般是有依赖关系的，有一定逻辑的。

与 C/C++，Java，Fortran，Pascal 等程序设计语言相比，shell 脚本语言掌握起来要相对容易。Linux shell 没有复杂和丰富的数据类型，虽然 Linux shell 包含分支、判断、循环等功能，但 shell 脚本在本质上的确就是将某些 Linux 命令“堆放”在一起（shell 脚本运行时，当中的命令依次接受脚本解释器的解释，例如，一个包含多条命令的脚本，即便最后一条命令有语法错误，前面的命令仍然可以照常运行；即便中间某条命令没有正常运行，假设后面的命令不依赖于它，后面的命令仍然可以照常运行）。所以很难看到逻辑非常复杂的 shell 脚本，看懂别人的 shell 脚本通常不是一件难事。

1.4　为什么要学习 Bash

第一个主流的 shell 是 Bourne shell，简称 sh，以发明者 Stephen Bourne（史蒂夫·波恩）的姓来命名。1979 年，sh 随着 UNIX 版本 7 发布并开始流行。

Bash 诞生于 1987 年，它是 sh 的超集，几乎完全兼容 sh 并拥有更多的特性和功能。在 UNIX 系统里，程序 sh 与 Bash 是不同的；而在 Linux 系统里，也有一个名字为 sh 的程序，

但它不是真正的 sh，而是 Bash，因为 Linux 里面的 sh 是一个指向程序 Bash 的符号链接，这个符号链接的设置是为了使 sh 脚本在 Linux 下不做修改就可运行。读者如果有机会看到较旧的脚本，会发现第一行大多是#!/bin/sh。有时，遇到年纪较大的工程师，会发现他在写 Linux 脚本的第一行时，仍然习惯写#!/bin/sh。

Csh 的语法有点类似 C 语言，这也是其名字叫 Csh 的原因。Tcsh 是 Csh 的超集。因为 Tcsh 兼容 Csh，所以多数用户选择功能更强的 Tcsh，使用 Csh 的人在逐渐减少，并且越来越多的 Linux 系统的 Csh 被设置为指向 Tcsh 的符号链接。Tcsh 也很重要，而且流传很广，但是 Tcsh 有个致命的弱点：没有函数功能。

Bash 是几乎所有 Linux 操作系统的默认的 shell。有时会发现某种 Linux 系统默认没有安装 Tcsh、Ksh 或 Zsh，但是，要发现某类 Linux 系统默认没有安装 Bash 是一件非常难的事情。Bash 简单易学，功能强大，并借鉴了 Ksh 和 Csh 很多有用的特性。如果掌握了 Bash，再要掌握其他种类的 shell，将会很容易。因此学习 Bash 是明智的选择。

1.5 Bash 学习环境的准备

1.5.1 Linux 的准备

细分 Linux 操作系统，有上百种之多。常用的 Linux 有 Redhat、Fedora、SUSE、Debian、Ubuntu 等。Linux 的多数命令，在各种 Linux 系统上没有本质差别。比较常用的命令及常用选项和参数，在各种 Linux 系统上几乎完全相同。如果某脚本在一种 Linux 系统上可以运行，在其他 Linux 系统上一般也可以正常运行，或者只需小小的改动即可运行。

如果读者已经有 Linux 服务器或 PC，并且知道了自己的账户名和密码，那么就已经有了 Bash 的学习环境，因为前面说过，Bash 是 Linux 系统的默认 shell。如果没有现成的 Linux 系统，可以自己试着安装和搭建。Linux 的安装盘不需要购买，上网任选一种 Linux，下载即可。以 Fedora 为例，http://fedoraproject.org/zh_CN/get-fedora#desktops 为下载链接。下载光盘镜像文件即 ISO 文件之后，可以将其制作成安装盘。制作安装盘的步骤和安装的步骤，在 get-fedora 网页上可以很容易找到。

可以将自己的计算机安装为 Linux 操作系统。或者将计算机安装为双系统，在启动的时候选择进入 Linux 或 Windows，对于只有一台计算机的用户来说，也许要频繁地重启计算机。总的来说，安装 Linux 比较容易，但对于初学者还是有一些难度。接下来的两小节提供了两种稍微简单的搭建 shell 学习环境的方法，并且可以使只有一台计算机的用户在 Windows 操作系统下“无重启”地学习 Linux shell 编程。

1.5.2 Cygwin 工具

Cygwin 是一个在 Windows 平台上运行的模拟 Linux/UNIX 环境的免费软件工具。它的下载、安装与使用，详见 http://www.cygwin.com/。

Cygwin 的下载和安装不难，大致过程为：先在 http://www.cygwin.com 下载一个名为 setup.exe 的可执行程序（目前新版本为 setup-x86.exe），运行该程序，然后单击 Next>按钮，得到 Cygwin 下载及安装的主界面，如图 1-2 和图 1-3 所示。



图 1-2　运行 Cygwin 的 Setup 程序



图 1-3　Cygwin 的下载及安装主界面

建议先下载到本地后再安装，就是下载的时候选择图 1-3 中的 Download Without Installing。Cygwin 不是一个很小的软件，下载和安装的时间也不是很短。除非网速很快，网络非常稳定，才可以选择 Install from Internet（从网络安装）。

下载的时候，可以选择默认下载，也可选择全部下载。如图 1-4 所示的多个“Default”就是默认下载，用鼠标单击最上面的 All@Default，所有的 Default 变为 Install 后，就是全部下载，如图 1-5 所示。下载完成后，再次运行 setup.exe，选择从本地安装，也就是选择图 1-3 中的 Install from Local Directory。安装包括默认安装和全部安装。

对于 Linux 下的各种应用不是很熟悉的人，如果硬盘空间足够大，在下载和安装的时候，最好选择全部下载和全部安装。全部下载和安装的用时较长，如果想快速熟悉 Cygwin 的下载、安装和基本使用，可以先选择默认下载和默认安装。

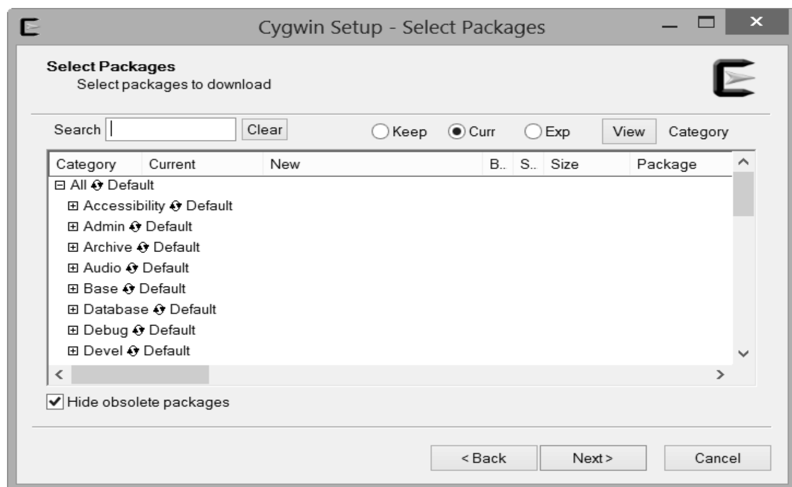


图 1-4　Cygwin 的默认下载

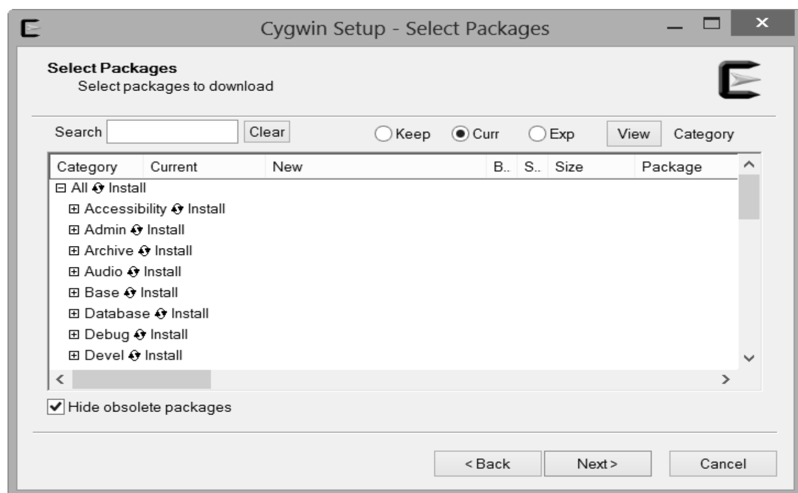
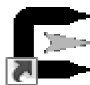


图 1-5　Cygwin 的全部下载

安装完成后，在桌面可以看到如下的图标：

双击该图标，即可进入 Cygwin 命令行字符界面（也叫 CLI，Command-Line Interface），会看到一个闪动的光标。这时所见的就是 Cygwin 默认的 shell 的“外表”，如下所示：

```
user@earth-PC ~
$
```

简单认识一下 Cygwin 界面：user 为账户名（account name），就是登录 Windows 的账号，earth-PC 为计算机名，$为命令行提示符。Cygwin 的默认 shell 为 bash，运行 echo $0 便可知。命令 echo 用来显示一行内容，或者显示变量的值；$0 是一个内置变量，表示 shell 的名字（如，bash、csh、tcsh 等）或者 shell 脚本的名字。

```
user@earth-PC ~
$ echo $0
bash
```

~表示用户的主目录，默认路径为/home/<account_name>，也称为宿主目录，也有人叫 home 目录或家目录。用 pwd 命令显示当前所在的路径（Print current/Working Directory）。

```
user@earth-PC ~
$ pwd
/home/user
```

如果将 Cygwin 安装在了 Windows 的 C:\cygwin 目录下，那么上面主目录/home/user 在 Windows 下的实际位置是 C:\cygwin\home\user。

注意，Cygwin 是一个类 UNIX/Linux 的在 Windows 下使用的工具。它不是 Linux 或 UNIX 操作系统。Cygwin 确实能完成很多原本在 Linux 下的工作，但它不能代替 Linux。它与 Linux 有一个明显的区别，Cygwin 默认不区分字母的大小写（像 Windows 一样，对文件名的字母大小写不予区分），Linux 是区分字母大小写的。某些在 Linux 下能正常编译的程序代码，到了 Cygwin 下则不能正常编译，除了编译环境不同之外，还有一个可能的原因就是 Cygwin 不区分字母大小写，例如，a.h 和 A.h 被认为是同一个文件。这一点，希望引起 Cygwin 新用户的注意。但 Cygwin 的使用者并没有因为这一问题而减少，毕竟多数人不会遇到这个问题，Cygwin 被广泛使用着。实际上，只要修改 Windows 与 Cygwin 的某些配置，Cygwin 就可以区分大小写了。本节不展开讨论这方面的内容，感兴趣的话可以查阅相关资料。

1.5.3　VMware 与 Linux 虚拟机

VMware 公司提供了一些软件工具，可以在安装了 Windows 的一台 PC 上虚拟出一台或者多台 Linux 计算机，它们可拥有各自的计算机名和 IP 地址，看上去像是两台或者多台计算机在同时工作。VMware Player 是其中的一个免费的软件，可登录 http://www.vmware.com/ 搜索并下载它，编写本章时较新的版本是 VMware Player V5.0.2 Build 1031769。VMware Player 可以在 Windows 平台上运行 Linux 虚拟机，当然，也可以在 Linux 平台上运行 Windows 虚拟机，甚至可以在某个版本的 Windows（或 Linux）上运行另一个版本的 Windows（或 Linux）虚拟机。

Linux 虚拟机的 Image 文件可以用 VMware 公司的工具生成，也可以直接从网上下载，或者复制别人做好的 Image 文件。复制别人的 Image 文件的时候，要问清楚账户名和密码，尤其是有管理员权限的账户 root 的密码。

Ubuntu 是一个以桌面应用为主的 Linux 操作系统。近几年，Ubuntu 越来越流行，尤其是随着 Google 公司的 Android 平台的兴起，使用和学习 Ubuntu 的人越来越多，因为基于 Android 的开发，大多是在 Ubuntu 操作系统上进行的。编写本书时较流行的版本是 12.04，可以在 http://www.trendsigma.net/vmware/或者 http://www.traffictool.net/vmware/找到 Ubuntu 虚拟机的 Image 的下载链接。下载解压后，可以看到一个 readme 文件，文件里面包含账户名和密码，以及 root 的密码。

下载 VMware Player 后，安装过程很简单，只需单击 Next 按钮即可，如图 1-6 所示。

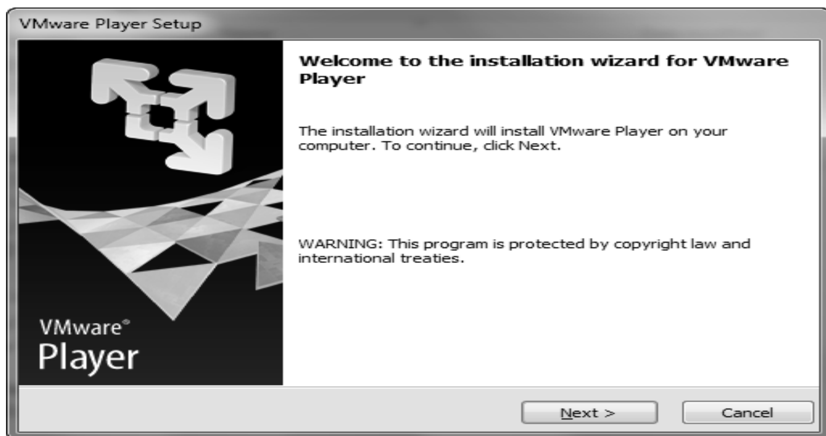


图 1-6　安装 VMware Player

安装完成后，运行 VMware Player，选择 Open a Virtual Machine，如图 1-7 所示。

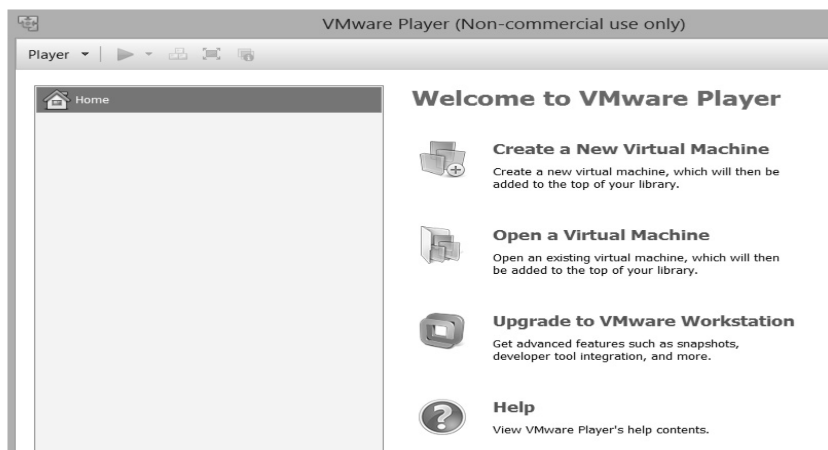


图 1-7　运行 VMware Player

找到 Ubuntu 虚拟机的文件 Ubuntu.vmx，将其打开，一台运行在 Windows 平台上安装了 Ubuntu 操作系统的虚拟计算机就展现在用户眼前，如图 1-8 所示。

如果想自己制作 Ubuntu 虚拟机的 Image 文件，可在 Ubuntu 网站下载光盘镜像文件，网址是http://www.ubuntu.com/download/desktop，如 ubuntu-12.04-desktop-i386.iso，即 Ubuntu 12.04 的 ISO 文件。运行 VMware Player，选择图 1-7 中的 Create a New Virtual Machine，按照提示，可以得到自己制作的 Ubuntu 虚拟机的 Image 文件。制作的过程中，会得到提示：设置账户名和密码。一定要记好密码。

进入 Ubuntu Desktop 12.04 后，单击左上角的 Dash home，会出现搜索栏，如图 1-9 所示。在搜索栏内输入 Terminal，然后单击找到的 Terminal（实际上它是可执行程序/usr/bin/gnome-terminal），即可进入 Ubuntu 的 shell 命令行窗口，如图 1-10 所示。

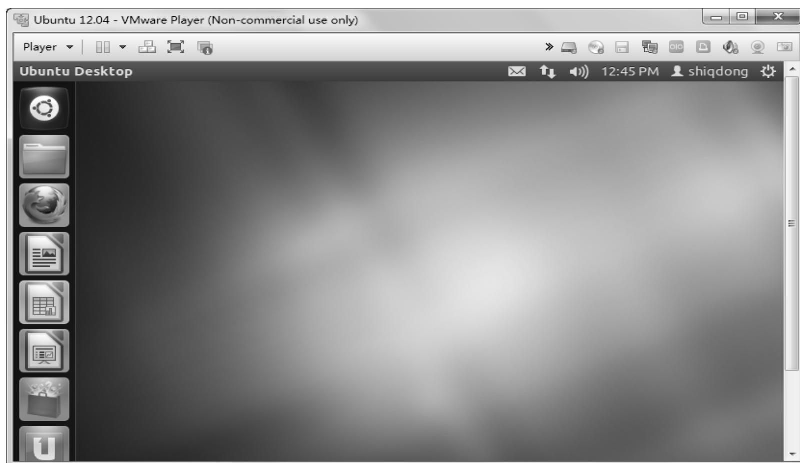


图 1-8　Ubuntu 12.04 虚拟机桌面

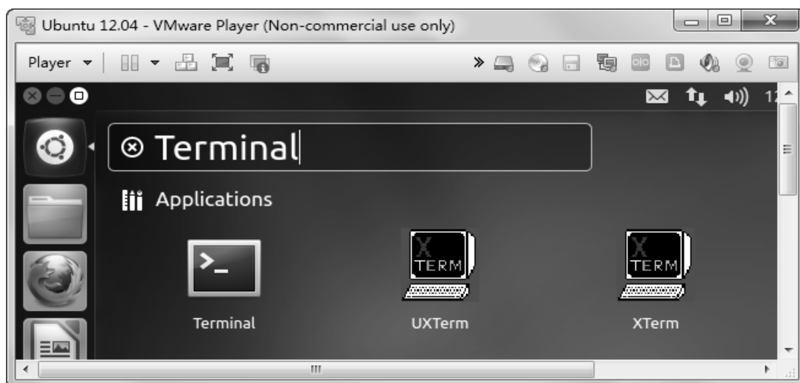


图 1-9　查找 Terminal

```
shiqdong@ubuntu: ~
shiqdong@ubuntu:~$ whoami
shiqdong
shiqdong@ubuntu:~$ hostname
ubuntu
shiqdong@ubuntu:~$ echo I study $0
I study bash
shiqdong@ubuntu:~$ pwd
/home/shiqdong
shiqdong@ubuntu:~$ uname
Linux
shiqdong@ubuntu:~$ uname -a
Linux ubuntu 3.8.0-29-generic #42~precise1-Ubu
ntu SMP Wed Aug 14 15:31:16 UTC 2013 i686 i686
 i386 GNU/Linux
shiqdong@ubuntu:~$
```

图 1-10　在 Ununtu 系统的 shell terminal 里输入几条命令

在图 1-10 中，展示了几条 Linux 命令的运行。whoami 的功能是显示当前账户名（回答 Who am I?的问题）；命令 hostname 用来显示计算机名；echo 命令中，内置变量$0 的值被代入，所以显示出 I study bash；pwd 命令显示了当前目录；uname 命令用来显示当前操作系统

名称，uname -a 显示系统详细信息。

如果计算机配置低，可以安装 Lubuntu。Lubuntu 是快速、轻量级的 Ubuntu 变体，它使用 LXDE 桌面（Light weight X11 Desktop Environment），占用的系统资源比较少。

在 Lubuntu 中进入命令行窗口的方法如图 1-11 所示。像在 Windows 中启动一个程序一样，选择 Start→Accessories→LXTerminal，即可进入命令行界面。另一种方法是在桌面单击右键，看见如图 1-12 所示的图形子菜单时，选择 Terminal 即可。

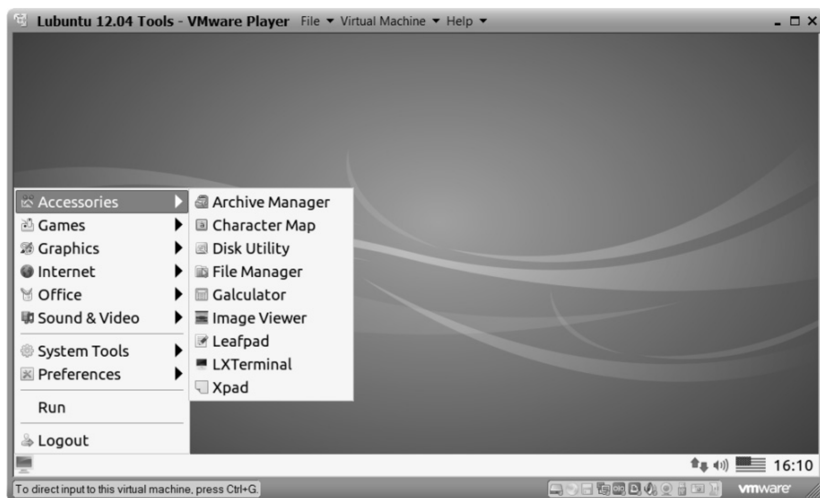


图 1-11　在 Lubuntu 12.04 中进入 Terminal



图 1-12　在 Lubuntu 12.04 中进入 Terminal（通过弹出菜单）

实际上，对于 Ubuntu 和 Lubuntu，都可以按〈Ctrl+Alt+T〉快捷键，以快捷方式打开一个 Terminal。

1.6　Linux 命令格式简介

Linux 命令的基本格式如下：

```
命令 [选项] [参数 1] [参数 2] …
```

执行命令时，最简单的情况是：只有命令名。例如，前面提到的 pwd、cal 和 date 等。绝大多数 Linux 命令带有选项和参数。在接下来的例子中，无论遇到的是命令还是命令加选项或参数，一般就叫做命令，或者叫做命令行。

选项（option）决定着命令的执行方式，决定着命令是否和如何“施展”其某方面的功能。单字符选项的前面为一个减号“-”（也叫短线或连字符，注意，不是下画线），多字符选项（也叫单词选项或者长选项）的前面为两个减号“--”。例如，在 C 语言中，转义字符\n 表示换行符，在 Linux 的 echo 命令中，\n 可以有同样的意义，这取决于用户使用什么选项。在 echo 命令中使用选项-e 时，表示使能（enable）转义，使用选项-E 时，表示禁止（disable）转义，默认为禁止转义。

下面的两条 echo 命令，参数都是“\nGood morning\n”。当使用选项-e 时，显示一空行、Good morning 和一空行：

```
$ echo -e "\nGood morning\n"
                              # 显示空行
Good morning                  # 显示早安
                              # 显示空行
```

如果使用选项-E，\n 原样显示出来，没有起到换行符的作用：

```
$ echo -E "\nGood morning\n"
\nGood morning\n
```

命令的单词选项，如--version 可用来查看命令的版本信息。以 date 命令为例：

```
$ date --version
date (GNU coreutils) 8.13
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Written by David MacKenzie.
```

1.7　如何获得 Linux 命令的帮助

Linux 下的命令非常多，大多数命令的参数和选项也很多，没有人能记住全部的命令、参数和选项。重要的不是全都记住，而是知道如何获得帮助。下面介绍获得帮助的两种常用方法。

第一种常用的查询某命令帮助的方法是，输入命令的名字，后跟单词选项--help。例如，运行 date --help，可以得到 date 命令的帮助，节选部分内容如下：

```
$ date --help
Usage: date [OPTION]... [+FORMAT]
  or:   date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]
Display the current time in the given FORMAT, or set the system date.
```

```
  -d, --date=STRING          display time described by STRING, not `now'
  -f, --file=DATEFILE        like --date once for each line of DATEFILE
  -r, --reference=FILE       display the last modification time of FILE
  -R, --rfc-2822             output date and time in RFC 2822 format.
                             Example: Mon, 07 Aug 2006 12:34:56 -0600
......
  -s, --set=STRING           set time described by STRING
  -u, --utc, --universal     print or set Coordinated Universal Time
      --help                 display this help and exit
      --version              output version information and exit

FORMAT controls the output.  Interpreted sequences are:
  %%     a literal %
  %a     locale's abbreviated weekday name (e.g., Sun)
  %A     locale's full weekday name (e.g., Sunday)
  %b     locale's abbreviated month name (e.g., Jan)
  %B     locale's full month name (e.g., January)
  %c     locale's date and time (e.g., Thu Mar  3 23:05:25 2005)
  %C     century; like %Y, except omit last two digits (e.g., 20)
  %d     day of month (e.g, 01)
  %D     date; same as %m/%d/%y
  %e     day of month, space padded; same as %_d
  %F     full date; same as %Y-%m-%d
  %g     last two digits of year of ISO week number (see %G)
......
```

一条看似简单的 date 命令，就有这么多的选项和参数，Linux 的命令大多是这样的，但一般只需要记住常用的选项和参数的使用方法即可。

在命令的语法当中，中括号内的选项或者参数为可选的（不是必选的），竖线表示或的意思。例如，在 date 命令的帮助信息中，可以看见 date [-u|--utc|--universal]，意思是使用选项-u、--utc、或者--universal 都是可以的，都是打印输出此刻的协调世界时（Coordinated Universal Time，普通人几乎不需要将时间精确到秒，可以认为它也是格林尼治时间）。例如：

```
$ date -u
Sat Aug 11 10:02:21 UTC 2012
```

另一种常用的查询命令帮助的方法是，输入 man，后面跟命令名字。man 是 manual（手册）的缩写。例如，运行 man pwd，得到 pwd 命令帮助信息如下：

```
NAME
       pwd - print name of current/working directory
SYNOPSIS
       pwd [OPTION]...
DESCRIPTION
       Print the full filename of the current working directory.
```

```
        -L, --logical
               use PWD from environment, even if it contains symlinks
        -P, --physical
               avoid all symlinks
        --help display this help and exit
        --version
               output version information and exit
        NOTE: your shell may have its own version of pwd, which usually supersedes the version
        described here. Please refer to your shell's documentation for details about the options it
        supports.
AUTHOR
        Written by Jim Meyering.
REPORTING BUGS
        Report pwd bugs to bug-coreutils@gnu.org
        GNU coreutils home page: <http://www.gnu.org/software/coreutils/>
        General help using GNU software: <http://www.gnu.org/gethelp/>
        Report pwd translation bugs to <http://translationproject.org/team/>
COPYRIGHT
        Copyright © 2010 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or
        later <http://gnu.org/licenses/gpl.html>.
        This is free software: you are free to change and redistribute it. There is NO WARRANTY, to
        the extent permitted by law.
SEE ALSO
        getcwd(3)
        The full documentation for pwd is maintained as a Texinfo manual. If the info and pwd
        programs are properly installed at your site, the command
               info coreutils 'pwd invocation'
        should give you access to the complete manual.
```

man 命令提供的帮助信息是分页显示的，按回车键显示下一行，按空格键显示下一页，按〈B〉键返回到（backwards）上一页，按〈Q〉键退出（quit）。有时运行 man command 时会遇到“No manual entry for command”的情况，这是因为用户的 Linux 没有安装相应的帮助信息，也可能因为命令的名字输错了。例如，本想查看 pwd 命令的帮助，不小心输入了 pwdd：

```
$ man pwdd
No manual entry for pwdd
```

以上两种查询命令帮助的方法，对于很多（并非所有）Linux 命令是有效的。对于 Bash 的内置命令，查询帮助的方法则不是这样的。第 3 章将介绍 Bash 内置命令及如何查询其帮助。

实际上，还有其他方法。例如，运行 info 加命令的名字，可查看命令的帮助，按空格键显示下一页，按〈Q〉键退出。例如，运行 info pwd，得到帮助信息如下：

```
`pwd' prints the name of the current directory.   Synopsis:

     pwd [OPTION]...
```

```
   The program accepts the following options.   Also see *note Common
options::.

`-L'
`--logical'
     If the contents of the environment variable `PWD' provide an
     absolute name of the current directory with no `.' or `..'
     components, but possibly with symbolic links, then output those
     contents.   Otherwise, fall back to default `-P' handling.

`-P'
`--physical'
     Print a fully resolved name for the current directory.   That is,
     all components of the printed name will be actual directory
     names--none will be symbolic links.

   If `-L' and `-P' are both given, the last one takes precedence.   If
neither option is given, then this implementation uses `-P' as the
default unless the `POSIXLY_CORRECT' environment variable is set.
```

运行 whatis 加命令名字，可以查看命令的简要描述。如：

```
$ whatis pwd
pwd (1)              - print name of current/working directory
$ whatis date
date (1)             - print or set the system date and time
```

本书对于提及的命令大多“点到为止”，读者要多上机试验，遇到问题可以先用本节介绍的方法寻求帮助，也可参考 Linux 命令大全之类的书，还可以上网搜索。如果在执行别人的脚本时遇到错误，可以在脚本代码中搜索错误关键词，对找到问题所在会有帮助。

开源的 Linux 欢迎每个人，如果有人写了一个能解决某问题的脚本或者软件工具，它的通用性强，使用的人越来越多，很多情况下甚至不得不用，那么，一段时间后，很可能它会成为 Linux 的一条命令。

第 2 章　Linux 基础知识与常用命令

Linux 命令很多，本章只涉及比较基本的命令，及其常用的选项、参数和常见的用法。本章尽可能地通过例子讲解命令，而不是只给出命令格式和其长长的选项和参数列表。想看列表时，可以用第 1 章介绍的方法随时得到。

2.1　Linux 的文件与目录结构

Linux 下的文件可以分为三类：普通文件、目录和特殊文件（如符号链接文件、设备文件）。Linux 下的文件名的最大长度为 256 个字符，文件名通常由字母、数字、下画线、小数点和减号组成。文件名中可以包含空格和特殊字符（如：*、？），但包含空格和特殊字符时容易出现麻烦。Linux 系统区分大小写，如文件 a.txt、A.txt、a.txT 和 a.tXt 是不同的文件。

Windows 以文件名后缀（文件的扩展名）来确定文件的类型，例如 exe 表示可执行文件，doc 表示 Word 文档，jpg 表示图片等。Linux 不使用扩展名识别文件的类型，而是根据文件头的内容来识别其类型。为了提高可读性，在 Linux 中可以使用扩展名，但这对 Linux 系统来说没有任何作用。

Windows 可以有多个根目录，每块逻辑盘分别有一个根目录，如 C:盘和 D:盘的根目录分别是 C:\和 D:\，而 Linux 系统只有一个根目录，用斜杠“/”表示。要注意 Linux 下的斜杠与 Windows 下的斜杠的“倾斜方向”不同，Linux 下的斜杠就叫斜杠或者斜线；Windows 下的，叫做反斜杠或者反斜线。

Windows 的软件工具安装在 C:\Program Files 目录中，Windows 的系统文件安装在 C:\Windows 目录中。Linux 也有固定的目录名和相应固定的用途。在 Linux 下，运行“cd /”和“ls”两条命令，可以看见根目录下的很多子目录。Linux 的主要系统目录及简单描述见表 2-1。

表 2-1　Linux 的主要系统目录及简单描述

目　录	描　述
/bin	最主要的可执行文件存放的地方，这些可执行文件大多是 Linux 系统里最常用的命令
/boot	包括内核和其他系统启动时使用的文件
/dev	存放设备文件。Linux 系统把所有的外设都看成是一个文件，对代表某设备的文件的操作就表示对该设备的操作
/etc	存放系统的配置文件。在该目录及子目录下有很多关于配置的文本文件，如后缀为 conf 的文件
/home	系统默认的普通用户的主目录为/home/<account_name>，保存用户文件，包括用户自己的配置文件、文档、数据等
/lib	包含许多由/bin 和/sbin 中的程序使用的共享库文件。/usr/lib 中包含更多用于用户程序的库文件

（续）

目　录	描　　述
/lost+found	在系统修复过程中恢复的文件
/mnt	文件系统挂载点。一般用于安装移动介质，其他文件系统的分区、网络共享文件系统或者任何可安装的文件系统
/opt	存放可选择安装的文件和程序。主要由第三方开发者用于安装和卸载他们的软件包
/proc	这个目录中的文件其实不是存放在磁盘上的，该目录的文件系统叫做内存映像文件系统，是系统内核的映像。该目录里面的文件是存放在系统内存里面，可以通过察看这些文件来了解系统的运行情况
/root	系统管理员、超级用户 root 的默认主目录
/sbin	存放可执行文件，这里的可执行文件主要供超级用户管理系统时使用，普通用户几乎没有权限执行其中的程序。/usr/sbin 中也包括许多系统命令
/tmp	临时目录，当系统重启时，该目录中的文件会被自动清空
/usr	包括与系统用户直接相关的文件和目录，一些主要的应用程序也保存在该目录下
/var	包含经常改变的文件，如假脱机（spool）目录、日志目录、锁文件、临时文件等

2.2　查看文件清单命令 ls

ls 命令用来列出（list）当前目录下包含的文件和子目录，它相当于 DOS 的 dir 命令。例如：

```
$ ls
bin    cronjob.txt    makefile    mkview.pl    mt.sh    sy1.pl
```

上面的例子说明当前目录下面有 6 个文件和目录，但哪些是文件，哪些是目录，这里看不出来（很多 Linux 系统，可以通过颜色来区分文件和目录）。ls 命令的选项-l 最常用，-l 是 long 的意思，用于列出详细信息。带选项-l 之后执行结果如下：

```
$ ls -l
total 24
drwxr-xr-x  2    shiqdong prjsrcmn    4096   Aug 15    19:29    bin
-rw-r--r--  1    shiqdong prjsrcmn      64   Aug 15    19:28    cronjob.txt
-r-xr-xr-x  1    shiqdong prjsrcmn    4253   Sep 21    2011     makefile
-rwxr-xr-x  1    shiqdong prjsrcmn    3044   Aug 15    19:28    mkview.pl
-rwxr-xr-x  1    shiqdong prjsrcmn     171   Mar 9     16:09    mt.sh
-rwxr-xr-x  1    shiqdong prjsrcmn     121   Dec 23    2011     sy1.pl
```

ls -l 列出的信息有七个部分。第一部分有 10 个字符，首字符表示类型，d 表示目录，-表示普通文件；第二部分，对于文件，表示硬链接数（参见 ln 命令），对于目录，则表示其所含的子目录数，包括隐藏目录；第三部分表示文件的所有者；第四部分表示其所属的组；第五部分表示大小，即字节数；第六部分为文件的修改时间；第七部分为文件名本身。

选项-a 或者--all 用来查看目录下所有的文件和目录。运行 ls -a，会看到更多的东西：

```
$ ls -a
.   ..   .cshrc   .vnc   bin   cronjob.txt   makefile   mkview.pl   mt.sh   sy1.pl
```

名字以“.”开头的文件或目录叫做隐藏文件或隐藏目录，如上例中的.cshrc 是隐藏文

件。在上面的例子中，还有两个特殊的东西，一个是“.”，另一个是“..”。“.”表示当前目录，“..”表示当前目录的父目录，即上一级目录。Linux 下的任何一个目录，一定包含这两个特殊的隐藏目录。读者在学习 cd 和 cp 命令时，对它们两个会有进一步的认识。

Linux 命令的很多选项可以组合使用，例如 ls 命令的选项-l 和-a 可以一起使用，将列出包括隐藏文件和隐藏目录在内的所有文件和目录的详细信息：

```
$ ls -l -a
total 52
drwxr-xr-x  4   shiqdong prjsrcmn    4096  Aug 20   14:25  .
drwxr-xr-x 43   root     root       16384  Aug 15   19:29  ..
-rwxr-x---  1   shiqdong prjsrcmn    1430  Aug 20   14:25  .cshrc
drwxr-xr-x  2   shiqdong prjsrcmn    4096  Aug 20   14:25  .vnc
drwxr-xr-x  2   shiqdong prjsrcmn    4096  Aug 15   19:29  bin
-rw-r--r--  1   shiqdong prjsrcmn      64  Aug 15   19:28  cronjob.txt
-r-xr-xr-x  1   shiqdong prjsrcmn    4253  Sep 21   20:11  makefile
-rwxr-xr-x  1   shiqdong prjsrcmn    3044  Aug 15   19:28  mkview.pl
-rwxr-xr-x  1   shiqdong prjsrcmn     171  Mar 9    16:09  mt.sh
-rwxr-xr-x  1   shiqdong prjsrcmn     121  Dec 23   20:11  sy1.pl
```

实际上，ls -l -a，ls -la 和 ls -al，这三条命令的执行结果是一致的。Linux 命令的多个选项一起使用时，用一个减号即可，选项的放置次序一般可以任意，用-la 和-al，效果相同。

ls 的选项-t 表示按照修改时间由新到旧列出文件。那么，ls -lat 的作用就是，按照修改时间由新到旧、列出包括隐藏文件和隐藏目录在内的所有文件和目录的详细信息。注意，Linux 命令的选项并非都可以随意组合，有些选项不能一起使用，有的选项依赖于其他选项。

如果只想查看某个文件的信息，运行“ls -l 文件名”：

```
$ ls -l cronjob.txt
-rw-r--r--  1   shiqdong prjsrcmn      64  Aug 15   19:28  cronjob.txt
```

运行“ls -l 目录名”可以看到该目录所含的文件和子目录的信息。例如，查看目录 bin：

```
$ ls -l bin
total 1324
-r-xr-xr-x  1 shiqdong prjsrcmn  1326988 Feb 21   2011 7za
-r-xr-xr-x  1 shiqdong prjsrcmn     9189 Feb 11   2011 drv_label_tools.bash
-rwxr-xr-x  1 shiqdong prjsrcmn     3044 Dec 22   2010 mkview.pl
-rwxr-xr-x  1 shiqdong prjsrcmn      413 Feb 15   2012 mount_vobs
-rw-r--r--  1 shiqdong prjsrcmn       53 Feb 18   2011 query_space.sh
-rwxr-xr-x  1 shiqdong prjsrcmn      456 Dec 22   2010 sy22.pl
```

查看目录本身的信息，需要加选项-d，或者--directory：

```
$ ls -l -d bin                  # 或运行 ls -ld bin
drwxr-xr-x  2 shiqdong prjsrcmn   4096  Aug 15   19:29   bin
```

可见，目录 bin 的信息的第二部分，是数值 2，表示 bin 里面含有 2 个目录。从 ls -l bin 的输出可见，bin 里面有 6 个文件，没有目录，确切地说，是没有“非隐藏目录”，用命令 ls -la bin 就能看全了。因为 Linux 所有的目录都包含“.”和“..”这两个特殊的隐藏目录，所以目录 bin 的信息的第二部分是数值 2。可以这样说，任何目录的信息第二部分数值一定不小于 2。

2.3 浏览文件命令 cat、more、less、head 和 tail

cat 命令可以查看文件的内容，例如，查看某台 Linux 计算机的文件/etc/passwd（文件较长，只列出了前 15 行）：

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
```

cat 命令是“一股脑”地显示文件的全部内容，文件较长时，用 cat 命令查看文件有些不方便。遇到长文件，最好使用 more 命令，它将长文件分页显示，每显示一页，more 命令就会等待用户的进一步指令。在用 more 分页显示文件时，用户常用的指令有：按回车键，显示下一行；按空格键，显示下一页；按〈B〉键，返回到上一页；按〈Q〉键，退出，不再查看余下的内容。

要想分页显示长文件，还可以用 less 命令，less 命令的使用与 more 命令比较相似。

head 命令用来查看文件开头的部分，默认显示文件的前 10 行：

```
$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
```

```
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
```

可以用选项-n 加参数 K 来显示文件的前 K 行，-n K 可以缩写为-K。例如，显示/etc/passwd 文件的前 5 行：

```
$ head -5 /etc/passwd                # 用 head -n 5 /etc/passwd 也可以
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
```

tail 命令与 head 命令呼应，它默认显示文件的最后 10 行，用-n K 或者-K 来显示文件的最后 K 行。tail 命令有一个比较常用的选项-f，使用它，可以看见文件尾部内容的增长或变化。当一个文件的内容在不断地变化时（如日志文件，调试脚本时，脚本打印的调试信息等），可以运行 tail -f filename，它会把 filename 里最尾部的内容显示在屏幕上，并且不断刷新，从而显示最新的文件内容。9.7 节有关于 tail -f 应用的例子。

2.4　文件统计命令 wc

wc 命令用来统计出文件的行数、单词数（word counts）和字节数等。下面，先查看文件 email.txt 的内容，再对其统计：

```
$ cat email.txt
Hello Jack,
I'm in China.
Mike
$ wc email.txt
  3   6   31   email.txt
```

wc 命令的运行结果表明 email.txt 有 3 行、6 个单词和 31 个字节。连续的非空白字符为一个单词（空白字符包括空格、〈Tab〉键、换行符等），那么显然 email.txt 有 6 个单词。email.txt 中的每个字母、空格和标点符号各占一个字节，数一数，似乎 email.txt 应该有 28 个字节。学过 C 语言的读者就会明白，每一行的末尾还有一个换行（new line）符\n，加起来就是 31 了。

使用 wc 命令的选项-l、-w、-c 时，分别显示文件的行数、单词数、字节数，例如：

```
$ wc -l email.txt
3 email.txt
$ wc -w email.txt
6 email.txt
$ wc -c email.txt
31 email.txt
```

选项-m 用来显示文件的字符数：

```
$ wc -m email.txt
31 email.txt
```

email.txt 的字符数与字节数是一致的。如果一个文件包含全角标点或者汉字，那么它的字符数与字节数将不一致。

2.5　改变当前工作目录命令 cd

cd 命令用于改变当前的工作路径（change working directory），或者说 cd 命令用于进入某目录。

先确定当前目录是什么：

```
$ pwd
/home/user/tmp
```

假设用 ls 命令查询，知道了当前目录下面有个子目录叫 bin，可以进入 bin：

```
$ cd bin
```

则当前路径变为了/home/user/tmp/bin：

```
$ pwd
/home/user/tmp/bin
```

..表示父目录，现在进入父目录：

```
$ cd ..
$ pwd
/home/user/tmp
```

../..表示父目录的父目录，现在试着进入上两级目录：

```
$ cd ../..
```

果然进入到了上两级目录：

```
$ pwd
/home
```

前面的例子，命令 cd 的参数都是相对路径，也可以使用绝对路径（从根目录开始的全路径）：

```
$ cd /home/user/tmp/bin
$ pwd
/home/user/tmp/bin
```

使用参数“-”可以回到刚才的路径。在前面的例子中，在进入/home/user/tmp/bin 之前，路径为/home，所以运行 cd -就回到了/home 目录：

```
$ cd -
$ pwd
/home
```

不带任何选项和参数的 cd 命令，表示进入当前账户的主目录。假设当前账户名为 user，主目录就是/home/user：

```
$ cd
$ pwd
/home/user
```

~<account_name>表示账户 account_name 的主目录，默认为/home/<account_name>。假设当前计算机上有个账户为 maggie，那么命令 cd ~maggie 表示进入该账户的主目录：

```
$ cd ~maggie
$ pwd
/home/maggie
```

~表示当前账户的主目录，那么 cd ~表示进入当前账户的主目录：

```
$ cd ~
$ pwd
/home/user
```

命令 cd ~与 cd 的效果一致，但在一个脚本里，cd ~的意义明显，可读性更好。

下面再讲解一下 cd 命令、相对路径和绝对路径。cd 命令以及与路径相关的命令都可以使用相对路径和绝对路径。假设当前目录为/home/user/tmp/bin，要进入到/home/user/doc/report 目录，有两种方式可以达到目的，如图 2-1 所示。

方式一，使用相对路径：

```
$ pwd
/home/user/tmp/bin              # 先确认当前工作目录
$ cd ../../doc/report           # 进入父目录的父目录的子目录 doc 的子目录 report
$ pwd
/home/user/doc/report           # 确认已经成功进入到/home/user/doc/report 目录
```

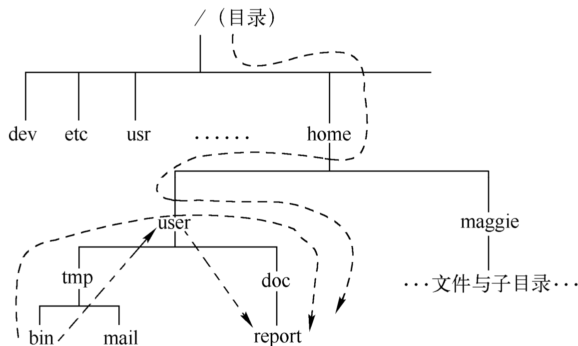


图 2-1　某目录层次示意图

命令 cd ../../doc/report 相当于图 2-1 中的从 bin 到达 report 的曲线。cd ../../doc/report 可以分成两步，cd ../..和 cd doc/report：

```
$ pwd
/home/user/tmp/bin              # 先确认当前工作目录
$ cd ../..                      # 进入当前目录的父目录的父目录
$ pwd
/home/user                      # 当前工作目录为/home/user
$ cd doc/report                 # 进入当前目录的子目录 doc 的子目录 report
$ pwd
/home/user/doc/report           # 确认已经成功进入到/home/user/doc/report 目录
```

命令 cd ../..相当于图 2-1 中的从 bin 到达 user 的直线，命令 cd doc/report 相当于从 user 到达 report 的直线。

方式二，使用绝对路径：

```
$ cd /home/user/doc/report      # cd 后面加全路径，则无需事先关心当前目录是什么
$ pwd
/home/user/doc/report           # 确认已经进入到/home/user/doc/report 目录
```

命令 cd /home/user/doc/report 相当于图 2-1 中的从根目录途经 home、user、doc，到达 report 的曲线。

两种路径都可以用，那么什么时候用绝对路径，什么时候用相对路径呢？这要视情况而定。执行命令时，使用哪种方便就使用哪种；写脚本时，使用哪种路径可以使得脚本的将来潜在的改动小、维护成本低、方便，就使用哪种路径。下面举个例子。

假设脚本 A 所处的位置固定不动，一直在/usr/bin 目录里面；假设目录 code 的位置“飘忽不定”，有时在/tmp 里面，有时在/home/user 里面，有时在其他的目录里（实际工作中，同一套代码，同一个软件工具，不同的人，不同的计算机，放置的位置可能不同，有时要求必须放置在事先确定好的位置，有时没有这样的要求）；假设脚本 B 在 code/config 目录里面，脚本 C 在 code/driver/cust 目录里面，如图 2-2 所示。因为目录 code 的位置不固定，所以脚本 B 和 C 所在的绝对路径是不确定的，但是 B 与 C 之间的相对位置是固定的。

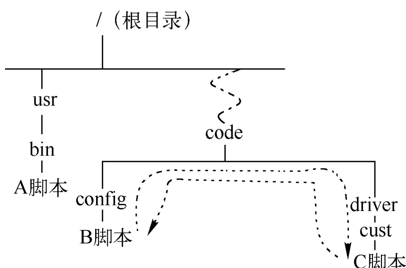


图 2-2　脚本互相访问（调用）的目录层次示意图

综上所述，脚本 B 和 C 调用脚本 A 时，要使用绝对路径：/usr/bin/A，因为 A 的位置是固定的；脚本 B 调用 C 时，可使用相对路径：../driver/cust/C（见图 2-2 从 B 到 C 的曲线）；脚本 C 调用 B 时，可使用相对路径：../../config/B（见图 2-2 从 C 到 B 的曲线）。这样的话，

无论 code 目录的位置在哪里，前面提到的三个调用都不会出现找不到被调用脚本的问题。

脚本 A 调用 B（或者 C）也是可能的。因为目录 code 的位置不固定，那么脚本 A 中调用 B 的命令是否需要改来改去呢？当 code 在/tmp 里面时，脚本 A 中调用 B 的命令为/tmp/code/config/B；当 code 在/home/user 里面时，脚本 A 中调用 B 的命令要改为/home/user/code/config/B。这样改来改去当然是可以的，但很麻烦。正确的做法是：不需要频繁修改脚本 A。在后面学习环境变量的时候，会“揭秘”答案。

2.6　创建目录命令 mkdir

mkdir 命令用来创建目录（make directories）。例如，创建一个名字为 document 的目录：

```
$ mkdir document
```

查看一下，可见目录创建成功：

```
$ ls -ld document
drwxr-xr-x 2 user user 4096 Aug 18 09:02 document
```

mkdir 命令可以同时创建两个或者多个目录：

```
$ mkdir email movie
$ ls -l
total 12
drwxrwxr-x 2 user user 4096 Aug 18 09:02 document
drwxrwxr-x 2 user user 4096 Aug 18 09:11 email
drwxrwxr-x 2 user user 4096 Aug 18 09:11 movie
```

假设在当前目录下需要创建一个 bin 目录，在 bin 目录下需要创建一个 test 目录，步骤如下：

```
$ mkdir bin                 # 创建目录 bin
$ cd bin                    # 进入目录 bin
$ pwd
/home/user/bin              # 现处在/home/user/bin 目录
$ mkdir test                # 创建目录 test
$ cd test                   # 进入目录 test
$ pwd
/home/user/bin/test         # 现处在/home/user/bin/test 目录
```

上面的步骤很多，假设要创建更深层的子目录，步骤会更多。mkdir 的选项-p 或者--parents 可以快速创建多层目录（parents 的作用是将各级目录的父目录及各级目录依次创建）：

```
$ pwd
/home/user                  # 确定当前目录是/home/user
$ mkdir -p bin/test/report  # “一口气”创建三层目录
$ cd bin/test/report        # 进入 bin/test/report
```

```
$ pwd
/home/user/bin/test/report              # 现处在/home/user/bin/test/report 目录
```

命令 mkdir -p 的参数可以是绝对路径：

```
$ mkdir -p /home/user/test1/test2/test3
$ cd /home/user/test1/test2/test3
$ pwd
/home/user/test1/test2/test3
```

可以用一张图来示意上面命令中的各级目录，如图 2-3 所示。命令 mkdir -p bin/test/report 对应图中左边的短曲线，命令 mkdir -p /home/user/test1/test2/test3 对应图中右边的长曲线。

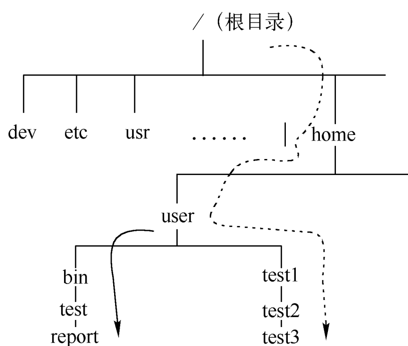


图 2-3　mkdir -p 命令示意图

mkdir -p 命令中的各级目录，可能都不存在，可能某一级或者某几级已经存在，也可能都已经存在，这几种情况，mkdir -p 命令都不会出错。从上层目录到最后一级目录，哪一级目录不存在就创建哪一级，如果都已经存在，相当于 mkdir -p 做了一个空操作。

2.7　复制命令 cp

cp 命令用于文件和目录的复制，也叫“拷贝”（copy），它的最基本的命令格式有两个参数，分别是“源”和“目的”。将 a.txt 复制（另存）为同一个目录下的 b.txt：

```
$ cp a.txt b.txt
```

如果将 a.txt 复制到/tmp 目录下，运行：

```
$ cp a.txt /tmp
```

如果将 a.txt 复制到/tmp 目录下，同时名字改为 e.txt，也就是将 a.txt 复制为/tmp 目录下的 e.txt，运行：

```
$ cp a.txt /tmp/e.txt
```

将 a.txt 复制到/tmp/doc 目录下，运行 cp a.txt /tmp/doc。当目录/tmp/doc 存在时，复制没

问题；当目录/tmp/doc 不存在（而/tmp 肯定存在）时，a.txt 被复制为/tmp 目录下的名字为 doc 的文件。

将其他目录下的一个叫做 report.txt 的文件复制到当前目录：

```
$ cp /home/maggie/document/report.txt .
```

将上面命令中的“.”换为“./”也可以，都表示当前目录。这个“.”不能省略，否则会出错（而在 DOS 下，copy 命令的第二个参数可以为空，为空时，表示复制到当前目录）。

将其他目录下的一个叫做 report.txt 的文件复制到另一个目录，例如：

```
$ cp /home/maggie/document/report.txt /home/mike/paper/
```

将多个文件复制到某个目录下，例如：

```
$ cp a.txt b.txt c.txt /home/maggie/document
```

将分布在不同的目录下的多个文件复制到某个目录下，例如：

```
$ cp ../mike/a.txt ../jack/b.txt /tmp/c.txt /home/maggie/document
```

命令 cp -r dir1 dir2 用来复制目录，将目录 dir1 里面的东西复制到目录 dir2。-r 代表 recursively，就是递归复制，cp 命令的选项-r 还可以换为-R 或者--recursive，这三种选项作用是相同的。dir1 里面可能不仅有文件，还有子目录，甚至有多层子目录，都会原封不动地复制到 dir2 之中。如果 dir2 不存在，则它会先被创建。

cp 命令是有危险的，可能会发生这样的事：文件 b.txt 存在，执行 cp a.txt b.txt 之后，原来的 b.txt 的内容消失了，被替换为 a.txt 的内容。cp 命令的选项-b 有备份作用（backup），使用该选项时，已经存在的目标文件的名字后会自动加波浪号~并得到备份。

例如，文件 a.txt 与 b.txt 都存在：

```
$ ls
a.txt   b.txt
```

用选项-b 时，b.txt 被备份为 b.txt~，a.txt 的内容覆盖 b.txt：

```
$ cp -b a.txt b.txt
$ ls
a.txt   b.txt   b.txt~
```

2.8　重命名或移动命令 mv

mv 命令的基本格式为 mv file1 file2，mv 是 move 的意思，file1 和 file2 分别是“源”和“目的”。源和目的，可以是文件与文件，可以是文件与目录，也可以是目录与目录，但不能是目录与文件。根据不同的情况，mv 命令有时做重命名的动作，有时做移动的动作。看几个例子，很容易理解这段话，无需死记硬背。

将文件 a.txt 重命名为 b.txt：

```
$ mv a.txt b.txt
```

将文件 a.c 移入 sourcecode 目录：

```
$ mv a.c sourcecode
```

🔔 注意：
将 a.c 移入 sourcecode 目录之前，要先确保目录 sourcecode 存在，否则，a.c 将被重命名为文件 sourcecode。

将三个文件移入到某个目录下，例如：

```
$ mv a.txt b.txt c.txt /tmp
```

将三个分布在不同的目录下的文件移入到某个目录下，例如：

```
$ mv ../mike/a.txt ../jack/b.txt /tmp/c.txt /home/maggie/document
```

创建目录 test1，然后将目录 test1 重命名为 test2：

```
$ mkdir test1
$ mv test1 test2
```

🔔 注意：
上面的这条命令可能会有两种结果：如果目录 test2 不存在，目录 test1 被重命名为 test2；如果目录 test2 存在，目录 test1 被移入 test2，test1 变为 test2 的子目录。

本节第一段说过，mv 命令的源与目的不能是目录与文件，否则，看看会怎样。

```
$ ls -ld cust net.log
drwxrwxr-x 2    user user      4096 Jul 30    09:32 cust
-rw-rw-r--  1    user user    278196 Dec 24    07:43 net.log
```

cust 是目录，net.log 是文件，运行下面的命令，得到“不能用目录覆盖非目录”的提示：

```
$ mv cust net.log
mv: cannot overwrite non-directory `net.log' with directory `cust'
```

这说明试图将目录重命名为文件（或者说，试图将目录移入文件）的操作是不允许的。

mv 命令也是有危险的，可能会发生这样的事：文件 a.txt 与 b.txt 都存在，执行 mv a.txt b.txt 之后，原来的 b.txt 的内容消失了，被替换为 a.txt 的内容。mv 命令的选项-b 有备份作用，与 cp 命令的选项-b 作用一致。

2.9　创建符号链接和硬链接命令 ln

Windows 下有快捷方式，Linux 下对应的是符号链接（symbolic links）。建立符号链接用命令 ln，它的命令格式为：

```
ln -s  目标文件(源文件)  链接文件
```

建立符号链接时，选项-s 或者--symbolic 是必须的。建立一个名字为 sym_a.txt、目标为文件 a.txt 的符号链接：

```
$ ln -s a.txt sym_a.txt
```

建立成功后，用 ls 命令查看：

```
$ ls -l *.txt
-rw-rw-r--    1         user user          96 Aug 18 22:20 a.txt
lrwxrwxrwx 1         user user            5 Aug 18 22:29 sym_a.txt -> a.txt
```

上例中，sym_a.txt 所在的那一行的第一个字符为小写字母 l，l 表示文件类型是符号链接。sym_a.txt 的字节数为 5，因为它的目标为 a.txt，5 实际上是字符串 a.txt 的长度。在命令 ln -s a.txt sym_a.txt 中，源文件 a.txt 如果不存在（这种情况叫做断链），命令也可以成功执行；或者运行完命令 ln -s a.txt sym_a.txt 之后，再把 a.txt 删除，也是允许的。有的 Linux 系统，若源文件不存在（断链时），符号链接文件的颜色为红闪（实际上，当用 ls -l 特别是 ls -lF 查看断链的符号链接文件时，有的 Linux 系统，其颜色是红闪，有的用不同的背景色来映衬；很多 Linux 系统对文件的不同类型、不同权限（后面很快要讲权限）等可以通过颜色来区分，如何区分、用什么颜色、或用什么背景色等都是可以配置的，本节不展开讨论这方面的内容）。

建立一个名字为 temp.txt、目标为 test 目录下面的文件 s.txt 的符号链接：

```
$ ln -s test/s.txt temp.txt
$ ls -l temp.txt
lrwxrwxrwx 1 user user 10 Aug 18 22:40 temp.txt -> test/s.txt
```

上例中 temp.txt 的字节数为 10，因为它的目标为 test/s.txt，10 就是字符串 test/s.txt 的长度。

也可以针对目录建立符号链接。例如，建立目标为目录/var/log/samba 的符号链接：

```
$ ln -s /var/log/samba samba_log
$ ls -l samba_log
lrwxrwxrwx 1 user user 14 Aug 18 22:48 samba_log -> /var/log/samba
```

删除一个符号链接文件，不影响它的目标文件。例如：

```
$ rm sym_a.txt
$ ls -l *.txt
-rw-rw-r-- 1           user user          96 Aug 18 22:20 a.txt
```

符号链接也叫软链接。ln 命令还可以用来建立硬链接，格式为：

```
ln  目标文件(已存在的源文件)  链接文件
```

硬链接是已存在文件的另一个名字。建立硬链接之前，“源”文件必须存在，否则会出错。这一点与符号链接不同。下面的命令中，如果 x.cpp 不存在，会出现 No such

file 的提示：

```
$ ln x.cpp y.cpp
ln: accessing `x.cpp': No such file or directory
```

下面举例说明硬链接的概念。假设当前目录下有一个文件 b.txt：

```
$ ls -l b.txt
-rw-rw-r-- 1 user user 18 Aug 20 12:14 b.txt
```

将 p_hard_link.txt 创建为 b.txt 的硬链接：

```
$ ln b.txt p_hard_link.txt
```

学习硬链接时，很多时候会与文件的复制搞混，为了对比硬链接和复制，将 b.txt 复制到 q_copy.txt，然后查看一下它们：

```
$ cp b.txt q_copy.txt
$ ls -l b.txt p_hard_link.txt q_copy.txt
-rw-rw-r-- 2 user user 18 Aug 20 12:14 b.txt
-rw-rw-r-- 2 user user 18 Aug 20 12:14 p_hard_link.txt
-rw-rw-r-- 1 user user 18 Aug 20 12:17 q_copy.txt
```

可见，b.txt 和 p_hard_link.txt 的硬链接数为 2（信息的第二部分的数值为 2），再创建一个 b.txt 的硬链接的话，这个数字将变为 3；而 q_copy.txt 的这个数字为 1。从上面的命令结果还可以发现，p_hard_link.txt 的信息的首字符为“-”而不是小写字母 l，表明它是普通文件。

查看一下它们的内容，是一模一样的：

```
$ cat b.txt
I like bash shell
$ cat p_hard_link.txt
I like bash shell
$ cat q_copy.txt
I like bash shell
```

ls 命令的选项-i 或者--inode 用来显示文件的 inode 号。这里不展开讲解 inode 号，简单地说，它像是一个不能重复的标号一样，每个文件或目录对应一个 inode 号。可见 b.txt 与 p_hard_link.txt 的 inode 号相同，而 q_copy.txt 的 inode 号与它们的不同：

```
$ ls -li b.txt p_hard_link.txt q_copy.txt
1978762 -rw-rw-r-- 2 user user 18 Aug 20 12:14 b.txt
1978762 -rw-rw-r-- 2 user user 18 Aug 20 12:14 p_hard_link.txt
1978764 -rw-rw-r-- 1 user user 18 Aug 20 12:17 q_copy.txt
```

修改 b.txt（或者 p_hard_link.txt）的内容，那么 p_hard_link.txt（或者 b.txt）也随之变化，它们的内容和修改时间总是保持一致的，实际上，它们互为硬链接。而 q_copy.txt 是对 b.txt 的简单复制，复制完成后，它们两个就“毫不相干”了，修改 q_copy.txt（或者 b.txt）

的内容，不会影响 b.txt（或者 q_copy.txt）。

下面修改 p_hard_link.txt，命令当中使用了输出重定向（后面将讲解重定向），意思是将 echo 命令显示的内容（引号中）存入文件 p_hard_link.txt 并覆盖原有的内容：

```
$ echo "I like bash shell script" > p_hard_link.txt
```

查看一下文件 p_hard_link.txt 的内容：

```
$ cat p_hard_link.txt
I like bash shell script
```

再查看 b.txt 的内容，与 p_hard_link.txt 的一致：

```
$ cat b.txt
I like bash shell script
```

查看一下它们三个，可见 b.txt 和 p_hard_link.txt 的字节数都变为了 25，修改时间都更新到 12:28 了（互为硬链接的文件是“联动”的），而 q_copy.txt 没有变化：

```
$ ls -l b.txt p_hard_link.txt q_copy.txt
-rw-rw-r-- 2 user user 25 Aug 20 12:28 b.txt
-rw-rw-r-- 2 user user 25 Aug 20 12:28 p_hard_link.txt
-rw-rw-r-- 1 user user 18 Aug 20 12:17 q_copy.txt
```

下面，先删除 b.txt，再查看 b.txt 和 p_hard_link.txt，可见，p_hard_link.txt 仍然存在，inode、字节数、修改时间等不变，只是硬链接数减少了 1：

```
$ rm b.txt                                              #  删除 b.txt
$ ls -li b.txt p_hard_link.txt
ls: cannot access b.txt: No such file or directory      # b.txt 不存在了
1978762 -rw-rw-r-- 1 user user 25 Aug 20 12:28 p_hard_link.txt
```

文件 p_hard_link.txt 的内容不变，未受到 b.txt 被删除的影响：

```
$ cat p_hard_link.txt
I like bash shell script
```

就是说，如果某文件不小心被删除了，而它的硬链接文件仍然存在的话，那么“源”的内容可以很容易地找回来。而符号链接，如果“源”不小心被彻底删除了，那么其内容几乎无法找回。如果希望两个或多个不一定在同一目录下的文件的内容总保持一致，并且希望相互备份，就可以使用硬链接。硬链接就是：一个文件，多个文件名，每删除一个链接文件则减少一个名字（链接数减少 1），当最后一个链接被删除后，文件才被删除。

硬链接有一个限制，链接在一起的文件必须处在同一个文件系统，否则，当链接它们时，ln 命令会出错，而符号链接没有这个限制（要知道当前的 Linux 有哪些文件系统，执行 df 命令，输出每一行的第一个字段是文件系统名）。

硬链接还有一些限制：有的 Linux 不允许建立目录的硬链接，有的 Linux 只允许超级账户（如 root）建立目录的硬链接。例如：

```
$ ln /tmp tmp_ln
ln: `/tmp': hard link not allowed for directory
```

2.10 显示当前目录命令 pwd

前面已经提到过用来显示当前工作路径的 pwd 命令。这里介绍两个有用的选项：-P 和-L。接着前面的例子，samba_log 是一个指向目录/var/log/samba 的符号链接：

```
$ ls -l samba_log
lrwxrwxrwx 1 user user 14 Aug 18 22:48 samba_log -> /var/log/samba
```

因为 samba_log 是指向目录而不是指向文件的符号链接，所以可以用 cd 命令进入 samba_log：

```
$ pwd
/home/user
$ cd samba_log
```

进入 samba_log 后，看看 pwd -L 和 pwd -P 分别显示什么：

```
$ pwd -L
/home/user/samba_log
$ pwd -P
/var/log/samba
```

pwd -P 显示的是实际路径，即物理路径；pwd -L 显示的是“表面上”的路径，即逻辑路径。

如果当前目录及其各级父目录都不是符号链接，那么当前的物理路径和逻辑路径是相同的，否则不同。

2.11 产生空文件或者改变文件时间戳命令 touch

运行 touch file 命令，如果 file 不存在，会产生一个空文件 file。例如，先确定 z.txt 不存在，再运行 touch z.txt，则会创建一个字节数为 0 的文件 z.txt：

```
$ ls -l z.txt
ls: cannot access z.txt: No such file or directory        # 先确定 z.txt 不存在
$ touch z.txt
$ ls -l z.txt
-rw-rw-r-- 1 user user 0 Aug 19 04:47 z.txt
```

无论 file 是否事先存在，运行 touch file 命令后，file 的修改时间均被设定为当前时间。下面的 k.txt 的修改时间与 date 命令显示的当前时间不同，运行 touch 命令后，k.txt 的修改时间变为了当前时间，即 15:12：

```
$ ls -l k.txt             # 确定 k.txt 已存在
```

```
-r--r--r-- 1 user user 0 Aug 19 09:29 k.txt
$ date +%T              #  显示当前时间
15:12:06
$ touch k.txt
$ ls -l k.txt           #  运行 touch 命令后，k.txt 的修改时间变为了当前时间 15:12
-r--r--r-- 1 user user 0 Aug 19 15:12 k.txt
```

touch 命令可以带上时间参数，将文件的修改时间设为特定时间，即某年某月某日的某一时刻。这里就不详细展开了，读者如果有兴趣可以查看帮助。

touch 命令可以改变文件的修改时间。其实，不带选项的 touch 命令将文件的访问时间、修改时间和状态改动时间都设定为了当前时间，用选项-a 设定访问（access）时间；用选项-m 设定修改（modification）时间。那么这三种时间各是什么意思呢？修改时间，顾名思义，就是文件最后一次被改动并保存的时间；访问时间，就是最后一次查看文件的时间，例如用 cat 命令查看它；状态改动时间，讲过 chmod 命令之后再解释它。

用 ls -l 看到的是修改时间，加上选项-u，即用 ls -lu 看到的是访问时间。下例中文件的修改时间和访问时间都是 11:29：

```
$ ls -l myscript.bash
-rw-rw-r-- 1 user user 29 Aug 15 11:29 myscript.bash
$ ls -lu myscript.bash
-rw-rw-r-- 1 user user 29 Aug 15 11:29 myscript.bash
```

下面使用 cat 命令“访问”一下它，而不要修改它：

```
$ cat myscript.bash
#!/bin/bash
echo I love bash
```

再查看它的信息，可见，修改时间没变，访问时间更新了：

```
$ ls -l myscript.bash
-rw-rw-r-- 1 user user 29 Aug 15 11:29 myscript.bash
$ ls -lu myscript.bash
-rw-rw-r-- 1 user user 29 Aug 19 16:30 myscript.bash
```

文件的三种时间当中，很容易想到，用户最关心并且最常用的是修改时间。Linux 下的某些命令和工具的执行，与文件的时间戳是相关的。

例如，有时候会用 find 命令搜索修改时间在某天之前（或者之后）的文件。如果对搜索结果想“操控”一下，可以事先使用 touch 命令。

再举个例子，假设有多人协同开发软件，每个人每天在 18:00 之前提交自己的经过单元测试的修改内容到代码库，每晚 23:00 有工具链可以自动编译最新的软件。工具先扫描文件的时间戳，如果发现所有的文件的修改时间都是昨日 18:00 以前的，说明当天没有人修改代码，则无需编译（使用前一次的编译结果即可）；如果发现当天只有一个或几个文件被修改，则只编译与其相关的模块即可，然后链接。Linux 下有个叫 make 的工具，可以做这样的事情。

2.12 查看账户名及其所属组的命令 whoami、id 和 groups

查看当前的账户名，除了前面提到的 whoami 命令，还有带选项-un 的 id 命令：

```
$ whoami
user
$ id -un
user
```

Linux 系统的每个账户都至少属于一个组，运行 groups 命令可查询当前账户属于哪些组：

```
$ groups
user adm dialout cdrom plugdev lpadmin admin sambashare
```

groups 命令输出中的第一个组叫做首要组，其他叫次要组。id 或者 id -a 命令也可以查询当前账户属于那些组，同时可查看账户号 uid 和各个组的组号 gid：

```
$ id -a
uid=1000(user) gid=1000(user) groups=1000(user), 4(adm), 20(dialout), 24(cdrom), 46(plugdev),
116(lpadmin), 118(admin), 124(sambashare)
```

要查询其他账户属于哪些组，带上账户名参数即可。例如，查询账户 maggie 属于哪些组，可运行 groups maggie 或者 id maggie。

2.13 文件与目录的权限

在某个目录，执行命令 ls -l 查看文件 makefile 结果如下：

```
$ ls -l makefile
-r-xr-xr-x   1       shiqdong prjsrcmn       4253   Sep 21   2011 makefile
```

ls 列出文件信息的第一部分后面的 9 个字符表示文件权限。r、w 和 x 分别表示读、写和执行权限，“-”表示无相应权限。第 2 个到第 4 个字符表示文件所有者的权限；第 5 个到第 7 个字符表示文件所属组的权限；第 8 个到第 10 个字符表示所有其他账户的权限。

例如，rw-r--r--，表示所有者对文件有读写权限，其他账户有读权限；rwxr-x---，表示所有者对文件有读写和执行权限，所属组的成员（以文件 makefile 为例，所属组成员是指所有属于组 prjsrcmn 的账户）有读和执行权限，其他账户则没有任何权限。

前面讲的一些命令，执行的时候有时候会因为权限不足而失败。例如，命令 cp file1 file2，如果 file2 已经存在，file1 将覆盖 file2，但不一定覆盖成功。下面的 z.txt 为一个只读文件（r--r--r--），将 a.txt 复制到 z.txt 时，会遇到权限不足（Permission denied）的提示：

```
$ ls -l a.txt z.txt
-rw-rw-r--   1 user user 15 2012-08-19 09:21 a.txt
-r--r--r--   1 user user 89 2012-08-19 09:22 z.txt
```

```
$ cp a.txt z.txt
cp: cannot create regular file `z.txt': Permission denied
```

用选项-f 或--force 就可以强行覆盖，且不会遇到权限不足的提示：

```
$ cp -f a.txt z.txt
$ ls -l a.txt z.txt
-rw-rw-r-- 1 user user 15 2012-08-19 09:21 a.txt
-rw-rw-r-- 1 user user 15 2012-08-19 09:48 z.txt
```

覆盖后，z.txt 与 a.txt 的内容相同，字节数一致，都是 15。用选项-i 或--interactive，在某文件被覆盖之前会得到提示，如：

```
$ cp -i a.txt z.txt
cp: overwrite `z.txt'?          # 这是系统给出的提示
```

这时，输入 y 或者 Y，按回车键，复制动作将继续；直接回车或者按〈Ctrl+C〉键，则取消该复制动作。

执行命令 mv file1 file2 时，也可能会因为权限不足而遇到与 cp 命令相同的情况，处理方法及选项-i 与-f 的使用方法与 cp 的完全相同。例如，将文件 a.c 移入目录 sourcecode 时，如果目录 sourcecode 中已经有同名文件，使用选项-i，会得到如下提示：

```
$ mv -i a.c sourcecode
mv: overwrite `sourcecode/a.c'?
```

总之，运行命令 cp 或者 mv 有一定风险（文件可能会被覆盖，原来的内容有可能丢失），如果事先确定命令没问题，可以使用选项-f 或--force；如果不确定，为稳妥起见，希望得到提示的话，使用选项-i 或--interactive。

对于一个文件而言，某账户若有了读权限，则意味着该账户可以查看它；有了写权限，就意味着该账户可以修改它，不使用-f 选项就可以覆盖或者删除它（见 2.17 节）；有了执行权限意味着该账户可以运行它（见 3.5 节）。

目录的权限的含义与文件的稍有不同，一个账户有了某目录的读权限时，该账户可以浏览它（如用 ls 命令查看目录中的文件列表）；有了写权限时，该账户可以在该目录中增加新文件，可以删除原有的文件（如果文件自身的权限也允许被删除）；有了执行权限时，该账户可以进入该目录。下面举例说明。查看以 t_dir 开头的几个目录，结果如下：

```
$ ls -ld t_dir*
d---------    2 user user 4096 Jan 2 07:07 t_dir0
d--x------    2 user user 4096 Jan 2 07:04 t_dir1
dr-x------    2 user user 4096 Jan 2 07:06 t_dir5
drwx------    2 user user 4096 Jan 2 07:23 t_dir7
```

输出结果中每一行的首字符均为 d，表明它们都是目录。账户 user 对于目录 t_dir0 没有任何权限，不能进入目录，不能查看目录：

```
$ cd t_dir0
bash: cd: t_dir0: Permission denied
$ ls -l t_dir0
ls: cannot open directory t_dir0: Permission denied
```

对于目录 t_dir1，账户 user 只有执行权限，可以进入目录，但不能查看目录：

```
$ cd t_dir1
$ pwd
/home/user/t_dir1
$ ls -l
ls: cannot open directory .: Permission denied
```

这时，账户好像进入了一间漆黑的屋子，不知道屋子多大，不知道屋里面有什么。很少有人给目录这样设置权限（仅仅设置目录的执行权限）。

对于目录 t_dir5，账户 user 有读和执行权限，可以进入目录，可以查看目录。因为没有写权限，所以不能在该目录增加新文件，也不能删除文件（不能增减文件）：

```
$ cd t_dir5
$ pwd
/home/user/t_dir5
$ ls
x.txt    y.txt    z.txt                # 可以查看目录中的文件列表
$ cp /tmp/a.txt ./                     # 不能增加文件（不能把其他目录的文件复制过来）
cp: cannot create regular file `./a.txt': Permission denied
$ rm z.txt                             # 不能减少文件（不能删除文件）
rm: cannot remove `z.txt': Permission denied
$ mv -f x.txt ww.txt                   # 不能更名（更名相当于增加一个文件，同时减少一个文件）
mv: cannot move `x.txt' to `ww.txt': Permission denied
```

对于目录 t_dir7，账户 user 有完全的掌控权。账户 user 可以进入目录并查看，也可以在目录中增减文件。

目录的权限设置比较常见的有：

1）rwxrwxr-x，表示某个目录对组内开放权限，对组外只提供查看的权利。

2）rwxr-x---，表示某个目录由其所有者控制，对组内开放查看权限，对组外不开放。

3）rwx------，如果某个目录存放的是所有者的“私密性”的东西，就会如此设置。查看 home 目录下面的隐藏目录，一般会看到权限为 rwx------的目录。运行 cd ~，再运行 ls -ld .* | grep ^drwx-（该命令用到了管道线|和 grep 命令，后面的章节会讲到），即可查看这样的目录。例如：

```
$ ls -ld .* | grep ^drwx-
drwx------    3 user user 4096 2011-04-29 06:32 .config
drwx------    4 user user 4096 2014-01-02 11:10 .gconf
drwx------    8 user user 4096 2011-08-13 06:26 .gnome2
drwx------    4 user user 4096 2013-11-13 07:12 .mozilla
drwx------    2 user user 4096 2014-01-02 11:09 .pulse
```

如果试着进入其他账户的“私密性”的目录，肯定会吃“闭门羹”。例如，账户 user 进入并查看账户 maggie 主目录里面的一个隐藏子目录：

```
$ cd ~maggie/.config/                    #  也可以改为 cd /home/maggie/.config
bash: cd: /home/maggie/.config/: Permission denied
$ ls -l ~maggie/.config/                 #  也可以改为 ls -l /home/maggie/.config
ls: cannot open directory /home/maggie/.config/: Permission denied
```

前面说过 touch 命令可以改变文件的时间，其实，并非百分之百可以，因为有时也会遇到权限不足的问题。实际上，很多命令与文件和目录的权限有关，这里就不一一举例子了。

本节列举了不同权限的文件和目录，权限是如何设置的呢？下一节将给出答案。

2.14　改变权限命令 chmod

chmod 命令用来改变文件和目录的权限，选项为：[ugoa][+-=][rwx-]，其中 u 表示文件的所有者（user），g 表示文件的所属组（group），o 表示除所有者和所属组组员以外的其他账户（other），a 表示所有账户（all），+、-、=分别表示对权限的增加、减少、定义。下面举例说明如何改变文件的权限。

下面的 example.txt 是一个所有账户对其没有权限的文件（提示：运行 touch example.txt 再运行 chmod 000 example.txt 可以得到这样的文件）：

```
$ ls -l example.txt
---------- 1 user user 0 Aug 21 18:33 example.txt
```

给文件的所有者增加读写权限：

```
$ chmod u+rw example.txt
$ ls -l example.txt
-rw------- 1 user user 0 Aug 21 18:33 example.txt
```

给其他账户增加读写和执行权限：

```
$ chmod o+rwx example.txt
$ ls -l example.txt
-rw----rwx 1 user user 0 Aug 21 18:33 example.txt
```

去掉其他账户的读写和执行权限：

```
$ chmod o-rwx example.txt
$ ls -l example.txt
-rw------- 1 user user 0 Aug 21 18:33 example.txt
```

将读权限赋给文件的所属组：

```
$ chmod g=r example.txt
$ ls -l example.txt
-rw-r----- 1 user user 0 Aug 21 18:33 example.txt
```

将读和执行权限赋给文件的所属组：

```
$ chmod g=rx example.txt
$ ls -l example.txt
-rw-r-x--- 1 user user 0 Aug 21 18:33 example.txt
```

赋给所有账户读写和执行权限：

```
$ chmod a=rwx example.txt
$ ls -l example.txt
-rwxrwxrwx 1 user user 0 Aug 21 18:33 example.txt
```

去掉所属组的所有权限，并去掉其他账户的所有权限：

```
$ chmod go=- example.txt
$ ls -l example.txt
-rwx------ 1 user user 0 Aug 21 18:33 example.txt
```

去掉所有账户的所有权限：

```
$ chmod a=- example.txt
$ ls -l example.txt
---------- 1 user user 0 Aug 21 18:33 example.txt
```

上面的命令用 chmod a= example.txt 或者 chmod = example.txt 也可以。

改变文件的权限，还有一种“数值”的方法。r、w、x、-可以分别用 4、2、1、0 表示，它们可以组成 0～7 这 8 种数值，分别表示不同的权限控制。例如，3=2+1 表示且仅表示-wx，7=4+2+1 表示且仅表示 rwx 等，见表 2-2。用 chmod ugo file 命令来控制 file 的权限，这里的 ugo 与前面的用法不同，u、g、o 可以取 0～7 的任一数值，分别表示 file 的所有者、所属组、其他账户的权限，下面举例说明。

表 2-2　数值及其所表示的权限

数　值	相应的权限
0	---
1	--x
2	-w-
3	-wx
4	r--
5	r-x
6	rw-
7	rwx

根据表 2-2，4 表示 r--，那么 444 表示 r--r--r--，将文件的读权限赋给所有账户，可运行：

```
$ chmod 444 example.txt
$ ls -l example.txt
-r--r--r-- 1 user user 0 Aug 21 18:33 example.txt
```

对照表 2-2，要赋予所有者读写和执行权限 rwx，赋予所属组读和执行权限 r-x，给其他账户只读权限 r--，可运行：

```
$ chmod 754 example.txt
$ ls -l example.txt
-rwxr-xr-- 1 user user 0 Aug 21 18:33 example.txt
```

要取消所有账户的所有权限：

```
$ chmod 000 example.txt
$ ls -l example.txt
---------- 1 user user 0 Aug 21 18:33 example.txt
```

上面的命令用 chmod 0 也可以，但是 chmod 000 可读性好。

前面介绍的都是针对某个文件的权限的修改，对某个目录自身的权限的修改，命令是一样的。如果对某个目录及目录中的所有文件和子目录一起进行权限修改，需要使用选项-R 或者--recursive。例如 chmod -R 775 dir 的作用是将目录 dir 本身、dir 内的文件和子目录、子目录内的文件、子目录的子目录等的权限都设置为 775。

chown 和 chgrp 命令也与文件权限的定制和修改有关，它们的作用分别是改变文件的所有者和所属组（change owner 和 change group），这里就不详细介绍了。

2.15　SUID 与 SGID 以及粘滞位

2.15.1　SUID

用 ls -l 命令查看文件或者目录的权限时，可以看见 r、w 和 x，在自己的 Linux 系统里面查找，还能看见其他字母。进入/usr/bin、/bin 或者/sbin 目录，运行 ls -l | grep '^...s'，能看到字母 s。例如，在 Ubuntu 的/bin 目录下，执行命令：

```
$ cd /bin
$ ls -l | grep '^...s'
-rwsr-xr-x 1 root root   26252  Mar    2  2012 fusermount
-rwsr-xr-x 1 root root   88760  Mar   30  2012 mount
-rwsr-xr-x 1 root root   34740  Nov    8  2011 ping
-rwsr-xr-x 1 root root   39116  Nov    8  2011 ping6
-rwsr-xr-x 1 root root   31116  Apr    8  2012 su
-rwsr-xr-x 1 root root   67720  Mar   30  2012 umount
```

可见账户 root 对这些文件的权限为 rws，而不是 rwx。在解释字母 s 的含义之前，先介绍命令 passwd。命令 passwd 用来修改账户密码，任何账户都可以使用它。运行命令，输入旧密码和新密码，新密码需要输入两次且必须一致，则可以成功修改密码。例如：

```
$ passwd
Changing password for user.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

修改密码之后，文件/etc/shadow 会被更新，因为加密之后的密码存在/etc/shadow 中。查看一下它的权限：

```
$ ls -l /etc/shadow
```

```
-rw-r----- 1 root shadow 993 Feb 19 10:07 /etc/shadow
```

从表面看，只有账户 root 才有权限修改更新这个文件，然而任何账户都可以使得/etc/shadow 的内容更新（因为任何账户都可以运行 passwd 来修改自己的密码）。这是如何实现的呢？查看一下可执行文件 passwd：

```
$ which passwd
/usr/bin/passwd
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 41284 Apr  8  2012 /usr/bin/passwd
```

在表示文件所有者的执行权限的地方可以看见 s，而不是 x，它就是 SUID，SUID 是 set uid 的缩写。SUID 仅对可执行的二进制文件起作用。SUID 的作用是：其他账户在执行时具有文件所有者的权限。下面以命令 passwd 为例来理解 SUID 的作用。

执行命令 passwd 就是执行/usr/bin/passwd，而文件/usr/bin/passwd 有 SUID 属性并且所有者是 root，那么任何账户执行 passwd 时，就临时具有了与账户 root 一样的权限（也可以认为临时变为了 root），加密的新密码就可以写入文件/etc/shadow 了。

SUID 的设置有两种方法。例如，有一个可执行的文件 a.sh：

```
$ ls -l a.sh
-rwxr-xr-x 1 user user 0 Feb 19 11:29 a.sh
```

对文件所有者（u）增加 SUID 属性（+s）：

```
$ chmod u+s a.sh
```

文件 a.sh 具有了 SUID 属性：

```
$ ls -l a.sh
-rwsr-xr-x 1 user user 0 Feb 19 11:29 a.sh
```

另一种方法是，使用数值的方法。通过前面章节的内容，似乎所谓的数值方法就是在 chmod 后面跟三位八进制数。其实 chmod 后面可以跟四位八进制数，最前面一位为 4 时，表示设置 SUID 属性。例如：

```
$ chmod 4755 a.sh
$ ls -l a.sh
-rwsr-xr-x 1 user user 0 Feb 19 11:29 a.sh
```

设置 SUID 要同时保证文件对于所有者有执行权限，如果没有，会怎样？先取消 a.sh 的对于所有者的执行权限：

```
$ chmod 655 a.sh
$ ls -l a.sh
-rw-r-xr-x 1 user user 0 Feb 19 11:29 a.sh
```

再设置 SUID：

```
$ chmod u+s a.sh
```

```
$ ls -l a.sh
-rwSr-xr-x 1 user user 0 Feb 19 11:29 a.sh
```

可以看见一个大写的字母 S 而不是小写的 s，这是不正常的。增加文件 a.sh 的所有者执行权限，就正常了：

```
$ chmod u+x a.sh
$ ls -l a.sh
-rwsr-xr-x 1 user user 0 Feb 19 11:29 a.sh
```

2.15.2　SGID

学习了 SUID 之后，SGID 就容易理解了，它是 set gid 的缩写。可以对一个可执行二进制文件设置 SGID。SGID 的作用是：其他账户在执行时具有文件所属组的权限。

例如，在 Ubuntu 的/usr/bin 目录下，执行命令 ls -l | grep '^......s'，可以看见一些具有 SGID 属性的可执行文件：

```
$ cd /usr/bin
$ ls -l | grep '^......s'
-rwsr-sr-x 1 daemon daemon    42800  Oct  25    2011 at
-rwxr-sr-x 1 root    tty        9728  Mar  31    2012 bsd-write
-rwxr-sr-x 1 root    shadow    45284  Apr   8    2012 chage
-rwxr-sr-x 1 root    crontab   34776  Apr   2    2012 crontab
-rwxr-sr-x 1 root    mail      13932  Oct  17    2011 dotlockfile
-rwxr-sr-x 1 root    shadow    18120  Apr   8    2012 expiry
-rwxr-sr-x 1 root    mail       9684  Oct  18    2011 mail-lock
-rwxr-sr-x 1 root    mail       9684  Oct  18    2011 mail-touchlock
-rwxr-sr-x 1 root    mail       9684  Oct  18    2011 mail-unlock
-rwxr-sr-x 1 root    mlocate   34432  Aug  17    2011 mlocate
-rwxr-sr-x 1 root    ssh      128416  Apr   2    2012 ssh-agent
-rwxr-sr-x 1 root    tty       18036  Mar  30    2012 wall
-rwsr-sr-x 1 root    root       9524  Mar  22    2012 X
```

以 crontab 命令为例（该命令用来设置周期性的指令，这里不展开讨论它），具体解释一下 SGID 的作用。

```
$ ls -l /usr/bin/crontab
-rwxr-sr-x 1 root crontab 34776 Apr   2   2012 /usr/bin/crontab
```

任何账户都可以运行 crontab 命令，/usr/bin/crontab 所属组为 crontab（组名与命令名相同），任何账户在执行 crontab 命令时，都临时具有了属于 crontab 组时所具有的权限。

还可以对一个目录设置 SGID，这时，SGID 的作用是：任何账户如果可以在该目录内建立新文件或者新的子目录，那么新建的文件或子目录的所属组与该目录的所属组保持一致。下面举例说明。

账户 user 创建目录/home/user/doc，并在目录中创建新文件 a：

```
$ whoami
```

```
user                      # 确保是账户 user
$ pwd
/home/user                # 确保 doc 目录建在/home/user 下面
$ mkdir doc
$ chmod 777 doc           # 目录 doc 权限为 rwxrwxrwx，这两条命令可合并为 mkdir -m 777 doc
$ cd doc
$ touch a                 # 创建新文件 a
```

现在需要使用另外一个账户，先介绍切换账户的方法。最容易想到的方法是退出 Linux，用另外的账户重新登录。若不退出 Linux，运行命令 su <账户名>或者 ssh <账户名>@<计算机名或 IP>也可以切换账户。这几种方法当然都需要知道相应的账户密码。例如：

```
$ su shiqdong
Password:
$ ssh shiqdong@ubuntu
shiqdong@ubuntu 's password:
```

假设现已切换为账户 shiqdong，该账户在目录/home/user/doc 中创建新文件 b：

```
$ whoami
shiqdong                  # 确保是账户 shiqdong
$ cd /home/user/doc
$ touch b                 # 创建新文件 b
$ ls -l
total 0
-rw-rw-r-- 1 user         user          0 Feb 19 13:08 a
-rw-rw-r-- 1 shiqdong     shiqdong      0 Feb 19 13:08 b
```

可见文件 a 的所属组为 user，文件 b 的所属组为 shiqdong。下面，账户 user 将目录/home/user/doc 加上 SGID 属性：

```
$ whoami
user                      # 确保是账户 user
$ cd /home/user
$ chmod g+s doc           # 学习了 SUID 后，很容易想到 SGID 对应的命令是这一条
$ ls -ld doc
drwxrwsrwx 2 user user 4096 Feb 19 13:08 doc             # 可见，组权限为 rws
```

账户 shiqdong 重新在目录/home/user/doc 中创建新文件 b：

```
$ whoami
shiqdong                  # 确保是账户 shiqdong
$ cd /home/user/doc
$ rm b                    # 先删除 b
$ touch b                 # 再创建 b
$ ls -l
total 0
-rw-rw-r-- 1 user         user 0 Feb 19 13:08 a
```

```
-rw-rw-r-- 1 shiqdong        user 0 Feb 19 13:14 b
```

可见，这时文件 a 和 b 的所属组都为 user，和 doc 目录的所属组一致。

给文件或者目录增加 SGID 属性除了用命令 chmod g+s 之外，还可以用数值的方法，chmod 后面跟四位八进制数，最前面一位为 2 时，表示设置 SGID 属性。例如：

```
$ mkdir photo
$ chmod 2777 photo            # 这两条命令可以合并为 mkdir -m 2777 photo
$ ls -ld photo
drwxrwsrwx 2 user user 4096 Feb 19 13:31 photo
```

与 SUID 同样道理，设置 SGID 属性要同时确保目录或者文件对于所属组有执行权限，如果没有，会看见大写的 S，例如：

```
$ chmod 2767 photo
$ ls -ld photo
drwxrwSrwx 2 user user 4096 Feb 19 13:50 photo
```

这是不正常的，增加所属组的执行权限，即运行 chmod g+x photo 之后，就正常了。

2.15.3　粘滞位

粘滞位也叫 SBIT，是 sticky bit 的缩写，SBIT 的属性只有目录可以设置。SBIT 可以理解为防删除位，它的作用是：在一个大家都有权限的目录下，某个账户不再可以随便删除别人的文件或目录了。下面举例说明。

普通账户运行命令 su，并输入管理员账户 root 的密码，可以切换为账户 root：

```
$ su
Password:
```

账户为 root 时，默认提示符为#，不是$。账户 root 建立了一个对所有账户开放权限的目录/tmp/picture：

```
# whoami
root                      # 确保是账户 root，这时，默认提示符为#，不是$
# cd /tmp
# mkdir picture           # 创建目录 picture，权限设置为 rwxrwxrwx
# chmod 777 picture       # 这两条命令可以合并为 mkdir -m 777 picture
```

账户 shiqdong 进入目录/tmp/picture，创建了两个文件 file_s1 和 file_s2：

```
$ whoami
shiqdong                  # 确保是账户 shiqdong
$ cd /tmp/picture
$ touch file_s1 file_s2
```

账户 user 进入目录/tmp/picture，创建了一个文件 file_user，然后账户 user 删除了账户 shiqdong 的一个文件 file_s1：

```
$ whoami
user                            # 确保是账户 user
$ cd /tmp/picture
$ touch file_user
$ rm -f file_s1                 # 账户 user 删除了账户 shiqdong 的文件 file_s1
$ ls -l file_*
-rw-rw-r-- 1 shiqdong    shiqdong    0 Feb 19 14:23 file_s2
-rw-rw-r-- 1 user        user        0 Feb 19 14:24 file_user
```

可见，在一个大家都有权限的目录/tmp/picture 下，某个账户可以随便删除别的账户的文件和子目录。

下面为目录 picture 增加 SBIT 属性，用命令 chmod o+t 或者 chmod +t 都可以：

```
# whoami
root
# cd /tmp
# chmod o+t picture
# ls -ld picture
drwxrwxrwt 2 root root 4096 Feb 19 14:23 picture
```

账户 user 试图删除账户 shiqdong 的文件 file_s2，结果失败：

```
$ whoami
user
$ cd /tmp/picture
$ rm -f file_s2
rm: cannot remove `file_s2': Operation not permitted      # 删除失败，粘滞位起作用了
```

增加 SBIT 属性还可以用数值的方法，chmod 后面跟四位八进制数，最前面一位为 1 时，表示设置 SBIT 属性。例如：

```
$ mkdir -m 1777 cloth
$ ls -ld cloth
drwxrwxrwt 2 user user 4096 Feb 19 14:44 cloth
```

与 SUID 和 SGID 同样道理，设置 SBIT 属性要先确保目录对于其他账户有执行权限，如果没有，会看见大写的 T，例如：

```
$ chmod 1776 cloth
$ ls -ld cloth
drwxrwxrwT 2 user user 4096 Feb 19 14:44 cloth
```

SUID、SGID 和 SBIT 可以同时设置。4、2、1 分别表示设置 SUID、SGID、SBIT，而 4、2 和 1 可以组成 1～7 之间的数值，例如，3=2+1，3 表示同时设置 SGID 和 SBIT；6=4+2，6 表示同时设置 SUID 和 SGID。例如：

```
$ chmod 3777 cloth
$ ls -ld cloth
```

```
drwxrwsrwt 2    user    user 4096 Feb 19 14:44 cloth
$ chmod 6777 a.sh
$ ls -l a.sh
-rwsrwsrwx 1    user    user                0 Feb 19 11:29 a.sh
```

2.16　查看文件的三种时间

查看文件的三种时间可以用 stat 命令，例如：

```
$ stat mkview.pl
  File: `mkview.pl'
  Size: 537                 Blocks: 8              IO Block: 4096     regular file
Device: 802h/2050d          Inode: 1308483         Links: 1
Access: (0644/-rw-r--r--)   Uid: ( 1000/      user)    Gid: ( 1000/      user)
Access: 2012-08-20 04:07:54.000000000 -0400
Modify: 2012-08-15 07:15:38.000000000 -0400
Change: 2012-08-15 07:15:38.000000000 -0400
```

上面的 Access、Modify 和 Change 分别表示访问时间、修改时间和状态改动时间。stat 命令用来查看文件或者文件系统的状态（status），用它可以查看文件的很多信息，这里不对 stat 命令展开讨论。

也可以用 ls -lu、ls -l、ls -lc 命令分别查看文件的访问时间、修改时间和状态改动时间，只是时间不如 stat 的精确：

```
$ ls -lu mkview.pl
-rw-r--r-- 1 user user 537 2012-08-20 04:07 mkview.pl
$ ls -l mkview.pl
-rw-r--r-- 1 user user 537 2012-08-15 07:15 mkview.pl
$ ls -lc mkview.pl
-rw-r--r-- 1 user user 537 2012-08-15 07:15 mkview.pl
```

当文件的权限、所有者、所属组或硬链接数发生改变时，即运行 chmod、chown、chgrp 或 ln 命令时，文件的状态改动时间将改变。当文件内容改变时，即修改时间变化时，状态改动时间也随之变化。

例如，给 mkview.pl 加上执行权限：

```
$ chmod +x mkview.pl
```

它的执行权限加上了，同时状态改动时间更新了：

```
$ ls -lc mkview.pl
-rwxr-xr-x 1 user user 537 2012-09-03 04:44 mkview.pl
```

而访问时间和修改时间不变：

```
$ ls -lu mkview.pl
-rwxr-xr-x 1 user user 537 2012-08-20 04:07 mkview.pl
```

```
$ ls -l mkview.pl
-rwxr-xr-x 1 user user 537 2012-08-15 07:15 mkview.pl
```

命令 stat --printf="%x\n%y\n%z\n"加文件名，用来显示文件的三种时间。其中，%x、%y和%z 分别对应访问时间、修改时间和状态改动时间，运行 man stat，看帮助文件就清楚了。下面用 stat 命令查看此刻文件 mkview.pl 的三种时间：

```
$ stat --printf="%x\n%y\n%z\n" mkview.pl
2012-08-20 04:07:54.000000000 -0400
2012-08-15 07:15:38.000000000 -0400
2012-09-03 04:44:17.000000000 -0400
```

2.17 删除命令 rm 与 rmdir

命令 rm file 用来删除（remove）文件 file。rm 命令可以一次删除多个文件：

```
$ rm a.txt b.txt c.log
```

删除只读文件时会遇到提示信息：

```
$ rm z.txt
rm: remove write-protected regular file `z.txt'?
```

如果确定删除它，输入 y 或者 Y，再按回车键，文件 z.txt 就被删除了；若不想删除它，可直接按回车键或按〈Ctrl+C〉键。如果已经确定删除它，并且不希望看见“只读/写保护”的提示，可以使用选项-f 或者--force，即 rm -f z.txt。

删除文件的时候，无论文件的权限是什么，希望总是得到提示的话，可使用选项-i 或者--interactive，例如：

```
$ rm -i qbasic.tar.gz
rm: remove regular file ` qbasic.tar.gz'?
```

同样，如果确定删除它，输入 y 或者 Y，再按回车键；若不想删除它，可直接按回车键或按〈Ctrl+C〉键。

rmdir 命令用来删除目录，但它只能用来删除空目录。如果删除非空目录，会遇到提示：

```
$ rmdir temp
rmdir: failed to remove `temp': Directory not empty
```

此时先将 temp 目录清空，再用命令 rmdir 就可以删除它。这样做比较麻烦，特别是temp 包含子目录，甚至包含子目录的子目录时，需要将目录从下至上一级一级地清空。实际上，使用带选项-r、或者-R、或者--recursive 的 rm 命令即可将非空目录直接删除：

```
$ rm -r temp
```

如果目录 temp 本身，或者其中的某个文件，或者某个子目录的权限为只读时，执行上

面的命令会遇到“只读/写保护”的提示。如果不希望看见“只读/写保护”的提示，加上选项-f，两个选项合并为-rf：

```
$ rm -rf temp
```

即使 temp 是个包含多级子目录的非空目录，用上面的命令也可将目录 temp 连同里面的内容一并删除。命令 rm -r <目录>和 rm -rf <目录>常用来删除目录，而命令 rmdir 在删除目录方面不是很常用。

2.18　编辑文件命令 vi

nedit、gedit、emacs 和 xemacs 都是 Linux 下常用的文本编辑器。但对某一种 Linux 发行版而言，这几种编辑器不一定都安装了。例如，试着运行 nedit test.txt 就知道 nedit 是否已经安装。在编辑文本方面，可以认为它们相当于 Windows 下的记事本，但论功能，这几种编辑器远比记事本强大，这里不详细讨论它们。

上面提到的几种编辑器是图形界面的编辑器，可以方便地进行全屏编辑。下面简单介绍一下字符界面的编辑器 vi 和 vim。它们是 Linux 的常用编辑器，很多 Linux 发行版都默认安装了 vi 和 vim。vi 和 vim 命令繁多，但是如果熟练掌握并使用灵活之后将会大大提高工作效率。vi 是 visual interface 的缩写，vim 是 vi improved 的意思，即增强版的 vi。一般地，vi 就够用，如果想使用代码加亮的话可以选择 vim。现在，有的 Linux 系统的 vi 就是 vim。接下来的内容，不讨论 vi 与 vim 的区别，只涉及它们的基本操作。

vi 有 3 个模式：命令模式（command mode）、插入模式（insert mode）和底行模式（last line mode）。下面使用 vi 产生新文件 test.txt，内容如下，只有两行：

```
#!/bin/bash
echo "I like bash."
```

运行 vi test.txt，进入命令模式，如图 2-4 所示。

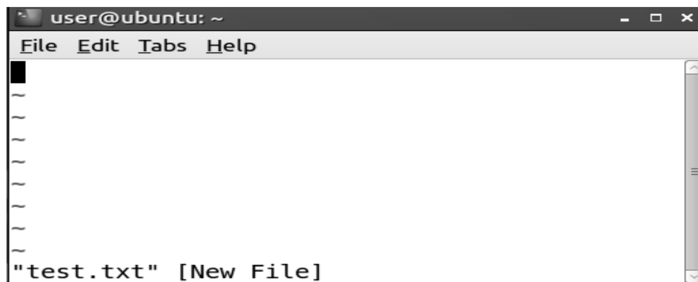


图 2-4　vi 的命令模式

这时，准备输入第一行内容，先输入“#”，发现键盘似乎不听从指挥。因为刚进入 vi 后，vi 处在命令模式，按〈I〉键，进入插入模式，则可以输入上面两行的内容。然后按键盘左上角的〈Esc〉键，回到命令模式，再按〈:〉键，进入底行模式，输入 wq，然后按回车键，即可存盘退出，如图 2-5 所示。

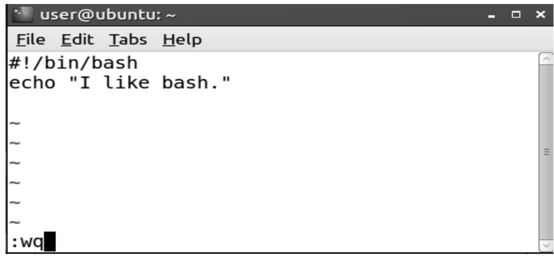


图 2-5　进入 vi 底行模式输入 wq 存盘退出

存盘退出后，查看一下文件的内容，可知，用 vi 编辑成功：

```
$ cat test.txt
#!/bin/bash
echo "I like bash."
```

有时把 vi 简化成两个模式，就是将底行模式也算入命令模式。保存文件和退出 vi 的命令见表 2-3。

表 2-3　保存文件和退出 vi 的命令

命　令	作　用
:w	保存文件，不退出 vi
:w readme.txt	保存至（另存为）readme.txt 文件
:q	退出 vi 编辑器（如果文件无修改）
:q!	退出 vi 编辑器，且放弃保存
:wq	退出 vi 编辑器，且保存文件

如果不是产生新文件，而是修改一个已有的文件，看看如何使用 vi。运行 vi test.txt，打开文件 test.txt，这时处在命令模式，先移动光标，找到需要修改的地方。按相应的键，光标移动方向如下：

〈h〉键向左、〈j〉键向下、〈k〉键向上、〈l〉键向右；空格键向右、〈Backspace〉键向左、〈Enter〉键移动到下一行首、〈-〉键移动到上一行首。

如果不是经常使用 vi，这比较难记。如果为了快速学会移动光标，只要先记住：使用键盘的上下左右方向键即可移动光标。

移动了光标、找到了需要修改的地方之后，可以修改文件。修改，包括删除字符或者行，还包括插入字符或者行。

删除字符或行、恢复删除（在命令模式下使用）的常用操作见表 2-4。

插入文本或行（在命令模式下使用）的常用操作见表 2-5。在 vi 命令模式下，执行表中的操作后将进入插入模式，按〈Esc〉键可退出插入模式回到命令模式。

表 2-4　vi 常用的删除与恢复删除操作

操　作	作　用
x	删除当前字符
nx	删除从光标开始的 n 个字符，n 为整数
dd	删除当前行
ndd	向下删除当前行在内的 n 行，n 为整数
u	撤销上一步操作
U	撤销对当前行的所有操作

表 2-5　vi 插入文本或行的操作

操　作	作　用
a	在当前光标位置的右边添加文本
i	在当前光标位置的左边添加文本
A	在当前行的末尾位置添加文本
I	在当前行的非空白字符的行首添加文本
O	在当前行的上面新建一行
o	在当前行的下面新建一行

下面修改文件 test.txt，把第二行改为 echo "We study bash."。运行 vi test.txt，用方向键将光标移到第二行的字符 I，按 6 次〈x〉键，可以把 I like 删除，按〈i〉键，进入插入模式，输入 We study，按键盘左上角的〈Esc〉键，再按一下〈:〉键，输入 wq 回车，存盘退出。查看一下文件的内容，可知，用 vi 编辑成功：

```
$ cat test.txt
#!/bin/bash
echo "We study bash."
```

用 vi 编辑 test.txt 时，在删除字符之后，可以试着按大写的〈U〉键，或者小写的〈u〉键，撤销刚才的删除。还可以试着按大写的〈O〉键，或者小写的〈o〉键，插入新行。

了解前面的内容后，就可以用命令 vi 创建新文件和修改老文件了。

vi 还有模式匹配及查找替换功能。下面介绍一个字符串替换的例子。有一个包含两行内容的文件如下：

```
$ cat blue.txt
I like the book, the blue book.
The book is blue.
```

假设要将 book 换为 magazine。运行 vi blue.txt，输入“:s/book/magazine”，按回车键，那么第一个 book 被改为 magazine，如图 2-6 所示。s 是 substitute（替换）的意思，第一个斜杠后面是被替换的内容，第二个斜杠后面是新的内容。

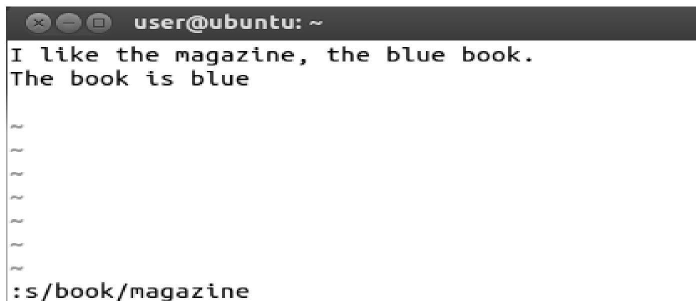


图 2-6　用 vi 对文件进行内容替换操作

再输入“:wq”，存盘退出，文件的内容被修改了：

```
$ cat blue.txt
I like the magazine, the blue book.
The book is blue.
```

可见，第一行的第二个 book 没有变，第二行的 book 也没有变。先将文件内容还原，如果希望全文的 book 都变为 magazine 的话，输入“:1,$s/book/magazine/g”，如图 2-7 所示，按回车键替换完成后，如图 2-8 所示。其中，1 表示第一行，$表示最末一行，标志 g 是 global 的意思，表示对一行从头至尾全部搜索替换，而不是只替换第一个匹配。

```
user@ubuntu: ~
I like the book, the blue book.
The book is blue

~
~
~
~
~
~
~
:1,$s/book/magazine/g
```

图 2-7 用 vi 对文件的全部内容搜索替换

图 2-8 的底行是对本次搜索替换的总结。再输入“:wq”，存盘退出，搜索替换全部完成。

```
user@ubuntu: ~
I like the magazine, the blue magazine.
The magazine is blue

~
~
~
~
~
~
~
3 substitutions on 2 lines
```

图 2-8 vi 对文件搜索替换的总结

本节介绍了 vi 的常用操作，但实际上，vi 的操作远不止这些。vi 功能较强，如果详细地讲解它的话，可以单独写一章。学习和使用 vi 的时候，要弄清楚当前处在什么模式，进入相应的模式（或者先回到相应的模式），再做相应的操作。发现某操作不起作用时，要确认一下当前处在什么模式，这是 vi 的初学者容易忽视之处。进入 vi、退出 vi、三种模式如何转换等，如图 2-9 所示，该图可以帮助读者学习 vi。

运行vi filename进入
命令模式
i/a/o
Esc
冒号
Esc
插入模式
底行模式
:wq
保存退出

图 2-9 vi 的三种模式

2.19 用于显示的命令 echo

命令 echo 用来显示参数，它的选项-e 和-E 在 1.6 节已经讲解，选项-e 比较常用。还有一个选项-n 比较常用，它的作用是去掉行尾的换行符（no trailing newline）。用 echo 显示参数时，行尾默认都是有换行符的。下面命令显示的 Morning 独占一行，后面的命令行提示符

在下一行，因为默认在 Morning 的后面有一个换行符：

```
$ echo Morning
Morning
$
```

有选项-n 时，下面命令显示的 Morning 和命令行提示符$在同一行，因为选项-n 使 Morning 的后面没有了换行符：

```
$ echo -n Morning
Morning$
```

当使用选项-e 时，一些反斜杠转义字符在 echo 命令里生效，其中的\n 在 1.6 节已经提到，见表 2-6。

表 2-6　echo 命令的转义字符

字符	含义
\a	响铃
\b	退格
\c	取消该字符之后的输出
\e	〈Esc〉键
\f	换页
\n	新行
\r	回车键
\t	〈Tab〉键（横向）
\v	竖向〈Tab〉键
\\	反斜杠本身
\0NNN	ASCII 码为一到三位八进制数对应的字符
\xHH	ASCII 码为一到两位十六进制数对应的字符

关于表 2-6 中的\c 需要解释一下。一些资料说它的作用是去掉行尾换行符，这样说不能算错，因为使用\c 时，\c 往往被放在行尾。例如，命令 echo -e "123\c"与 echo -n "123"的作用是相同的。但是，确切地说，\c 的作用是取消该字符之后的输出，包括行尾换行符。不把\c 放在行尾，试一试：

```
$ echo -e "123\c456789"
123$                              # 因为没有换行符，123 和命令提示符在同一行
```

可知，\c 之后的内容都不再显示，包括行尾换行符。

下面是几个 ASCII 码字符的例子（见表 2-6 的最后两行）：

```
$ echo -e "\050"
(
$ echo -e "\0175"
}
$ echo -e "\x26" "\x7b"
& {
```

2.20　查看文件类型命令 file

2.1 节提到，Linux 系统不用文件扩展名来确定文件类型。用户可以使用命令 file 查看文件的类型。例如，文件/bin/ls 是二进制可执行文件：

```
$ file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared
libs), for GNU/Linux 2.6.24, BuildID[sha1]=0x5f580e4b387193789cb865afdebb75442e1d5516,
stripped
```

b.zip 是压缩文件，c.txt 是文本文件：

```
$ file b.zip
b.zip: Zip archive data, at least v1.0 to extract
$ file c.txt
c.txt: ASCII text
```

first.sh（见 3.5 节）是 Bourne-Again shell（Bash）可执行脚本：

```
$ file first.sh
first.sh: Bourne-Again shell script, ASCII text executable
```

2.21　显示树状目录信息命令 tree

当目录层级比较多时，用 ls 命令不太方便，这时可以使用 tree 命令。查看某一个目录的树状信息用命令 tree <目录>。查看当前目录，不用跟目录名。例如：

```
$ tree
.
|-- chap03
|   |-- echo_test.sh
|   |-- exec_echo_test.sh
|   |-- first.sh
|   |-- from_win.sh
|   `-- z.txt
`-- chap04
    |-- current_directory.sh
    |-- display_age.sh
    |-- env_dollar_0.sh
    |-- env_process_id.sh
    `-- need.sh

2 directories, 10 files
```

希望只显示各级子目录（不显示当中的文件）时，用选项-d：

```
$ tree -d
.
|-- chap03
`-- chap04

2 directories
```

选项-p 用于显示类型和权限（permission），-s 用于显示大小（size），-D 用于显示文件的修改时间（moDification time）。这几个选项可以一起使用，例如：

```
$ tree -psD
.
|-- [drwxr-xr-x         4096 Jan   7 17:59]   chap03
```

```
|     |-- [-rwxr-xr-x        54 Jan     7 17:58]  echo_test.sh
|     |-- [-rwxr-xr-x        64 Jan     7 17:58]  exec_echo_test.sh
|     |-- [-rwxr-xr-x        78 Jan     7 17:58]  first.sh
|     |-- [-rwxr-xr-x        40 Jan     7 17:58]  from_win.sh
|     `-- [-rwxr-xr-x        33 Jan     7 17:58]  z.txt
`-- [drwxr-xr-x            4096 Jan     7 17:58]  chap04
      |-- [-rwxr-xr-x        77 Jan     7 17:58]  current_directory.sh
      |-- [-rwxr-xr-x        28 Jan     7 17:58]  display_age.sh
      |-- [-rwxr-xr-x        40 Jan     7 17:58]  env_dollar_0.sh
      |-- [-rwxr-xr-x        34 Jan     7 17:58]  env_process_id.sh
      `-- [-rwxr-xr-x        77 Jan     7 17:58]  need.sh

2 directories, 10 files
```

当目录的层级很多而又不需要查看太深层级时，可以使用选项-L 加层级数（level）。例如，需要在目录树中最深显示到第 3 层子目录，可运行命令 tree -L 3。

2.22　查找命令 find

命令 find 的常用格式为：

```
find [起始目录] 选项及寻找条件 [操作]
```

命令 find 默认为递归查找，从起始目录开始的每一层子目录都会进入查找。当没有指定起始目录时，默认从当前目录开始查找。默认的操作是-print，就是将找到的内容显示在屏幕上。命令 find 常用的选项见表 2−7。

表 2−7　find 命令常用选项

选项及参数	含　义
-name filename	按照名字 filename 查找，可使用通配符（如，*代表多个字符，?代表一个字符）
-perm mode	按照权限 mode 查找
-user account_name	查找所有者为某个账户的文件，-user 后面跟账户号 uid 也可以
-group group_name	查找所属组为某个组的文件，-group 后面跟组号 gid 也可以
-mtime {-n|+n}	用-n（+n）时，查找修改时间距现在 n 天之内（之前）的文件
-type {b|d|c|p|l|f}	查找某一类型的文件, b、d、c、p、l 和 f 分别表示块设备、目录、字符设备、管道、符号链接、普通文件
-newer file	查找修改时间比某个文件新的文件，! -newer 表示取反

例如，从当前目录下开始查找后缀名为.txt 的文件：

```
$ find . -name "*.txt" -print     # 可以改为 find -name "*.txt"
./a.txt
./script/chap06/name_list.txt
./script/chap03/z.txt
```

命令 find /tmp/code -name "test1.c" -mtime +3 -user maggie -perm 755 -type f 用来从

/tmp/code 目录里面开始，查找名字为 test1.c、最后一次修改时间在 3 天以上、所有者为账户 maggie、权限为 755 的普通文件（不是符号链接或其他类型）。

命令 find -newer a.txt 查找修改时间比 a.txt 更新的文件，!表示条件取反，find ! -newer a.txt 查找比 a.txt 更老的文件或者修改时间与 a.txt 相同的文件。!的取反功能可用在其他查找条件上。例如，命令 find ! -group smartguy 查找不属于组 smartguy 的文件。

命令 find 找到文件后的操作，除了有默认的-print，常用的还有“-exec 命令 {} \;”，{}代表找到的文件，结尾为\;，注意}与\之间要有空格。例如，查找 script 目录里面的后缀名为.txt 的文件，并用 ls -l 命令列出：

```
$ find script -name "*.txt" -exec ls -l {} \;
-rwxr-xr-x 1 shiqdong prjsrcmn 21 Dec 16 16:41 script/chap06/name_list.txt
-rwxr-xr-x 1 shiqdong prjsrcmn 33 Dec 16 15:16 script/chap03/z.txt
```

-exec 可以换为-ok，使用-ok 时，需要用户对操作确认。例如，接上面的例子，将找到的文件删除：

```
$ find script -name "*.txt" -ok rm {} \;
< rm ... script/chap06/name_list.txt > ? n                # 键盘输入 n，不删除
< rm ... script/chap03/z.txt > ? y                        # 键盘输入 y，删除
```

得到提示时，输入 n，不删除文件，输入 y，则文件被删除。

还可以使用管道及 xargs 命令对查找到的文件进行操作，这里不展开讲解 xargs 命令，它主要用于给其他命令传递参数，功能较强。例如，接上面的例子，将找到的文件的权限改为 700：

```
$ find script -name "*.txt" | xargs chmod 700
$ find script -name "*.txt" | xargs ls -l
-rwx------ 1 shiqdong prjsrcmn 21 Dec 16 16:41 script/chap06/name_list.txt
```

第 3 章　Bash 内置命令与环境简介

从本章开始正式进入 Bash 的世界。一个人到了新的生活环境，想要快速融入的话，一定要首先了解当地的风俗习惯、常用语和基本法规等，学习 Bash 也是一样的。本章将先简单介绍 Bash 内置命令和获得帮助的方法，然后介绍 Bash 的基本环境，并展开讲解常用的几条内置命令。

3.1　内置命令与外部命令

内置命令，简单地说，就是 Bash 自带的命令，也叫内建（built-in）命令。要查看内置命令的列表，运行命令 help 即可。help 本身就是一条内置命令。下面是在 Ubuntu 系统下的运行结果的部分内容：

```
$ help
GNU bash, version 4.2.24(1)-release (i686-pc-linux-gnu)
These shell commands are defined internally.  Type `help' to see this list.
Type `help name' to find out more about the function `name'.
Use `info bash' to find out more about the shell in general.
Use `man -k' or `info' to find out more about commands not in this list.

A star (*) next to a name means that the command is disabled.

 job_spec [&]                              history [-c] [-d offset] [n] or histo>
 (( expression ))                          if COMMANDS; then COMMANDS; [ elif CO>
 . filename [arguments]                    jobs [-lnprs] [jobspec ...] or jobs ->
……
 caller [expr]                             pwd [-LP]
 case WORD in [PATTERN [| PATTERN]...) >   read [-ers] [-a array] [-d delim] [-i>
 cd [-L|[-P [-e]]] [dir]                   readarray [-n count] [-O origin] [-s >
……
 disown [-h] [-ar] [jobspec ...]           test [expr]
 echo [-neE] [arg ...]                     time [-p] pipeline
 enable [-a] [-dnps] [-f filename] [nam>   times
……
 getopts optstring name [arg]              wait [id]
 hash [-lr] [-p pathname] [-dt] [name .>   while COMMANDS; do COMMANDS; done
 help [-dms] [pattern ...]                 { COMMANDS ; }
```

前面提到过的 cd、echo 和 pwd 出现在上面的列表里，表明它们是 Bash 的内置

命令。要查看某一条内置命令的帮助，运行 help 加该内置命令即可。例如，查看 cd 的帮助：

```
$ help cd
cd: cd [-L|[-P [-e]]] [dir]
    Change the shell working directory.

    Change the current directory to DIR.  The default DIR is the value of the
    HOME shell variable.

    The variable CDPATH defines the search path for the directory containing
    DIR.  Alternative directory names in CDPATH are separated by a colon (:).
    A null directory name is the same as the current directory.  If DIR begins
    with a slash (/), then CDPATH is not used.

    If the directory is not found, and the shell option `cdable_vars' is set,
    the word is assumed to be  a variable name.  If that variable has a value,
    its value is used for DIR.

    Options:
        -L        force symbolic links to be followed
        -P        use the physical directory structure without following symbolic
    links
        -e        if the -P option is supplied, and the current working directory
    cannot be determined successfully, exit with a non-zero status

    The default is to follow symbolic links, as if `-L' were specified.

    Exit Status:
    Returns 0 if the directory is changed, and if $PWD is set successfully when
    -P is used; non-zero otherwise.
```

help 命令有三个选项，选项-d 用于显示命令的简要功能介绍（short description），例如：

```
$ help -d pwd
pwd - Print the name of the current working directory.
```

使用选项-s，将显示命令的简要的语法格式（short usage synopsis），例如：

```
$ help -s pwd
pwd: pwd [-LP]
```

使用选项-m，将显示命令的手册页（manual page）形式的用法解释，例如：

```
$ help -m pwd
```

```
NAME
    pwd - Print the name of the current working directory.

SYNOPSIS
    pwd [-LP]

DESCRIPTION
    Print the name of the current working directory.

    Options:
      -L   print the value of $PWD if it names the current working
           directory
      -P   print the physical directory, without any symbolic links

    By default, `pwd' behaves as if `-L' were specified.

    Exit Status:
    Returns 0 unless an invalid option is given or the current directory
    cannot be read.

SEE ALSO
    bash(1)

IMPLEMENTATION
    GNU bash, version 4.2.10(1)-release (i686-pc-linux-gnu)
    Copyright (C) 2011 Free Software Foundation, Inc.
    License GPLv3+: GNU GPL version 3 or later http://gnu.org/licenses/gpl.html
```

前面提到，运行 help 可以得到 Bash 内置命令列表，准确地说，是 Bash 的内置命令和关键字（keyword）的列表。例如，for、while、until、do 和 done 等，是组成循环的关键字；if、then、elif、else 和 fi 等，是组成条件判断的关键字。如果将这些关键字称为内置命令是不完全准确的，这一点，读者在学习循环和判断命令的章节之后会有所体会。关键字也叫保留字（reserved word）。查询关键字的帮助信息，同样是 help 加上它的名字，例如：

```
$ help while
while: while COMMANDS; do COMMANDS; done
    Execute commands as long as a test succeeds.

    Expand and execute COMMANDS as long as the final command in the
    `while' COMMANDS has an exit status of zero.

    Exit Status:
Returns the status of the last command executed.
```

Bash 主要内置命令的简介见表 3-1。

表 3-1　Bash 主要内置命令简介

命　令	摘　　要	命　令	摘　　要
.（点号）	dot 命令，也叫点命令，读取文件并于当前 shell 中执行它	jobs	列出后台作业
:（冒号）	空操作，null 命令，返回状态总是成功（退出状态总是 0）	kill	向由 PID 号或作业号指定的进程发送信号，运行 kill -l 查看信号列表
alias	显示或者创建命令的别名	let	用来计算算术表达式的值，并把结果赋给变量
bg	把作业放到后台	local	用在函数中，定义局部变量
bind	显示或设置键盘按键及其相关的功能	logout	退出登录 shell
break	从 for、while 和 until 等循环中退出	mapfile	从标准输入为数组赋值，或将文本文件读入数组
builtin	执行指定的内置命令	popd	从目录栈中删除目录
case	保留字，多路条件结构	printf	格式化打印
cd	改变工作目录	pushd	向目录栈中增加目录
command	跳过函数和别名检查（与命令同名的函数和别名会被忽略），运行命令	pwd	打印出当前的工作目录
compgen	产生可能的完整的命令	read	从标准输入中读取一行
complete	规定命令补齐如何进行	readarray	同 mapfile 命令
continue	跳过 for、while 和 until 的本次循环，进入下一轮循环	readonly	将变量设为只读
declare	声明变量并赋予其属性，也可用来显示变量的属性和值	return	从函数返回
dirs	显示当前记录的目录（即目录栈，pushd 的结果）	select	保留字，建立选择菜单
disown	从作业表中移除作业	set	设置选项和位置参数
echo	显示参数	shift	左移位置参数（脚本参数或者函数参数）
enable	启用或禁用内置命令	source	同“.”命令，读取文件并于当前 shell 中执行它
eval	将参数读入 shell，并执行产生的命令	suspend	挂起当前 shell 的执行
exec	运行命令，替换当前 shell	test	判断条件表达式，退出状态为 0（1）表示条件成立（不成立）
exit	退出 shell	times	显示由当前 shell 启动的进程运行所累计用户时间和系统时间
export	导出变量，使变量可被子 shell 识别，创建环境变量时常用	trap	设置信号捕获过程
false	什么也不做，总返回失败结果，退出状态总是 1	true	总返回成功结果，退出状态总是 0
fc	列出历史命令或修改并重新执行历史命令列表中的一部分	type	显示关于命令类型的信息
fg	把作业放到前台	typeset	同 declare 命令，已过时但仍能使用，应尽可能多用 declare 命令
for	保留字，循环结构	ulimit	显示或设置进程可用资源的最大限额
function	定义函数	umask	显示或设置创建新文件时的权限掩码
getopts	解析处理命令行选项及参数	unalias	取消别名设置
hash	确定并记忆完整路径名，控制用于加速命令查找的内部散列表	unset	取消变量或函数定义
help	显示内置命令的帮助信息	until	保留字，直到型循环结构
history	显示命令历史	wait	等待进程或作业结束，并报告它的结束状态
if	保留字，条件判断结构	while	保留字，当型循环结构

与内置命令相对的是外部命令，外部命令不是 Bash 自带的，而是安装在 Linux 系统某个地方，是“看得见、摸得着”的命令。例如，ls 命令不在内置命令的列表里面，使用 which 命令可以查询到：

```
$ which ls
/bin/ls
```

再看看 vi 命令在哪里：

```
$ which vi
/usr/bin/vi
```

不同的 Linux 系统里，有些命令的具体安装位置可能会不一样。另外，有些命令既是内置命令也是外部命令，或者说 Linux 安装了与 Bash 内置命令同名的某些外部命令，例如 echo 就是这样的命令。在内置命令列表里可以看见 echo，在外部命令中也能找到它：

```
$ which echo
/bin/echo
```

如果运行了一条 echo 命令，是内置的 echo 执行了呢，还是外部的 echo 执行了呢？答案是内置的 echo 执行了，因为内置命令的优先级高于外部命令（有关命令的优先顺序，见 3.6.1 节）。如果带上外部命令的全路径，那么外部命令得到执行，例如：

```
$ echo "Today is Sunday"
Today is Sunday                     # 内置 echo 命令的执行结果
$ /bin/echo "Today is Sunday"       # 带上全路径的 echo 命令
Today is Sunday                     # 外部 echo 命令的执行结果
```

第 1 章介绍了查看命令帮助的方法，实际上说的是查看外部命令帮助的方法。运行 echo --help 时，由于使用的是内置 echo 命令，运行的结果是打印出了字符串--help，而没有看到它的帮助信息：

```
$ echo --help
--help                              # 内置 echo 命令显示了字符串--help
```

带上外部 echo 命令的全路径，可以查看到外部 echo 命令的帮助：

```
$ /bin/echo --help
Usage: /bin/echo [SHORT-OPTION]... [STRING]...
  or:  /bin/echo LONG-OPTION
Echo the STRING(s) to standard output.

  -n             do not output the trailing newline
  -e             enable interpretation of backslash escapes
  -E             disable interpretation of backslash escapes (default)
      --help     display this help and exit
      --version  output version information and exit

If -e is in effect, the following sequences are recognized:
```

```
\\          backslash
\a          alert (BEL)
\b          backspace
\c          produce no further output
\e          escape
\f          form feed
\n          new line
\r          carriage return
\t          horizontal tab
\v          vertical tab
\0NNN       byte with octal value NNN (1 to 3 digits)
\xHH        byte with hexadecimal value HH (1 to 2 digits)

NOTE: your shell may have its own version of echo, which usually supersedes
the version described here.  Please refer to your shell's documentation
for details about the options it supports.

Report echo bugs to bug-coreutils@gnu.org
GNU coreutils home page: <http://www.gnu.org/software/coreutils/>
General help using GNU software: <http://www.gnu.org/gethelp/>
For complete documentation, run: info coreutils 'echo invocation'
```

既然存在内置命令与外部命令同名的情况，那么同名的它们在用法上有区别吗？就常用的选项和参数而言几乎没有差别，至少是没有大的区别。把内置 echo 命令的帮助信息（运行 help echo），与外部 echo 命令的帮助信息（运行/bin/echo --help）对比一下，可以发现，差别很小。类似地，pwd 是 Bash 的内置命令，同时，Linux 系统也安装了外部的 pwd：

```
$ which pwd
/bin/pwd
```

同样，运行 help pwd，再运行/bin/pwd --help，对比一下，就知道，内置命令 pwd 与外部命令/bin/pwd 的主要选项-L 和-P 的功能没有差别。外部命令/bin/pwd 的长选项--logical 和--physical 分别相当于-L 和-P，内置 pwd 命令不支持这两个长选项。

3.2 认识 Bash 环境

“环境”，可以说它是看不到的，也是看得到的。所谓环境是指保证一个系统能正常运行的必备条件。

例如，使用中国移动的手机号，在马路上直接拨打 10086，可以接通中国移动客服中心，但是到了一个很深的矿井里则无法拨通 10086。这是因为，在马路上，人们处在中国移动信号覆盖的环境之中，在矿井里，没有这样的环境。因此中国移动信号就是成功拨打 10086 必备的条件。

同样的一个脚本，有时候在一台 Linux 计算机上可以正常运行，在另一台上则不能正常

运行；或者，某人（即使用某个 Linux 账户）可以正常运行该脚本，另一个人（即使用另一个账户）却不能正常运行。原因是两台 Linux 计算机的环境不同，或者两个账户的环境不同。

3.2.1　命令行提示 PS1

在前面的例子中，为简单起见，命令行提示符均为$。实际上，命令行提示很少是单个的字符$。在第 1 章，曾经见到过 Bash 命令行提示，形式为：“账户名@计算机:当前路径$”。其实，这只是命令行提示的某一种“外表”，用户是可以定制它的。

PS1 为 Bash 的内置变量，用于设置第一命令提示，也叫主命令提示。为什么叫第一？因为还有 PS2、PS3 和 PS4 等与命令提示有关的内置变量（在后面章节会讲到）。设置 PS1 时，\h 表示计算机名，\t 表示当前时间，\u 表示账户名，\w 表示当前路径（主目录显示为~）。这几个是比较常用的，还有很多其他选项，见表 3-2。

表 3-2　Bash 命令提示符内置环境变量设置

设　置	描　述
\d	日期，格式为 weekday month date，例如：Wed Sep 05，即 9 月 5 日周三
\H	完整的主机名称。例如，ubuntu.linux.com
\h	仅取主机的第一个名字，如上例，为 ubuntu
\s	shell 名称，即 bash
\t	显示时间为 24 小时格式：HH:MM:SS，例如，16:48:55
\T	显示时间为 12 小时格式：HH:MM:SS，例如，04:48:55
\A	显示时间为 24 小时格式：HH:MM，例如 16:48
\@	显示时间为 AM/PM 格式：HH:MM AM/PM，例如 04:48 PM，下午 4 点 48 分
\u	当前的账户名称
\v	Bash 的版本信息，例如 4.2
\V	带有修正版本的 Bash 版本信息，例如 4.2.10
\w	完整的工作目录名称，主目录会以~代替
\W	只列出最后一级目录，而不是完整的工作目录
\#	当前命令窗口中下达的命令序号，从 1 开始
\!	历史命令数目
\$	提示字符，用户如果是 root，则提示符为# ，普通用户则为$

下面将命令行提示变为“计算机名#当前时间>”，Linux 系统将显示按下回车键那一刻的时间：

```
user@ubuntu:~$ PS1="\h#\t>"
ubuntu#09:41:21>
ubuntu#09:41:33>                    # 按回车键，可见时间在前进
```

下面将命令行提示变为“计算机名@$”，然后再变回老样子，即“账户名@计算机名:当前路径$”：

```
ubuntu#09:41:34> PS1="\h@\$ "
```

```
ubuntu@$ PS1="\u@\h:\w\$ "
user@ubuntu:~$
```

PS1 不一定必须按照表 3-2 中的选项进行设置，它可以等于任何字符串、任何变量（包括内置变量）、任何命令或者函数的返回值，虽然实际上很少有人这样用，但只要用户喜欢就可以。下面将命令行提示变为“China>”：

```
user@ubuntu:~$ PS1="China>"
China>
```

如果现有的命令行提示不是自己喜欢的，可以给 PS1 赋值，设定自己喜欢的提示。以后进入任何 Bash shell，遇到没见过的提示并且想知道是如何设置的，都可以执行 echo $PS1 命令。执行 echo $PS1 时，如果遇到 PS1: Undefined variable 的提示信息，说明用户的 shell 不是 Bash，一定是其他种类的 shell。PS1 是 Bash 的有默认值的内置变量，不会是没有定义的。

3.2.2 搜索路径 PATH

可能会有人问，用户运行 ls，系统如何知道运行/bin 目录下面的 ls 呢？运行 vi，系统如何知道运行/usr/bin 目录下面的 vi 呢？执行 which echo，系统如何知道应该列出/bin/echo 呢？这是由 Linux 的搜索路径决定的，查看内置环境变量 PATH 即可知道当前的搜索路径：

```
$ echo $PATH
/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

PATH 中的目录以冒号间隔开，可见/bin 和/usr/bin 都在搜索路径之中。运行某条命令时，系统按照 PATH 中目录的先后顺序，依次在这些目录中搜索该命令，搜索不到的话，系统将提示：command not found（命令未发现）；一旦搜索到，将执行该命令，并停止对后面路径的搜索。假设有两个名字相同、处在不同目录的命令，那么处在搜索路径前面的将得到执行机会。

PATH 是可以改变的，例如，将目录/tmp/bin 添加到搜索路径的后面：

```
$ PATH="$PATH:/tmp/bin"
```

查看一下，添加成功了：

```
$ echo $PATH
/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/tmp/bin
```

也可以重新定义 PATH，甚至 PATH 中的目录可以是不存在的目录：

```
$ PATH="/abc/def:/xyz/mn"
$ echo $PATH
/abc/def:/xyz/mn
```

现在 PATH 被定义为两个不存在的目录，这时，执行 ls，看看会发生什么结果：

```
$ ls
```

```
Command 'ls' is available in '/bin/ls'
The command could not be located because '/bin' is not included in the PATH environment variable.
ls: command not found
```

系统已经找不到 ls 命令了，因为 PATH 被定义为两个不存在的目录。看来搜索路径 PATH 的定义与修改不可随意。如果已经将 PATH 改坏了怎么办？可以参考其他账户的 PATH 值，把自己的改回为原来的值。如果忘记原来的值，最简单的解决办法是重新打开一个命令窗口，关闭原来的窗口。实际上，如果不小心将任何一个环境变量改坏了，重新打开一个 shell terminal 是最简单最有效的解决方法。

防止环境变量被改坏的稳妥方法，就是每次修改之前先备份。例如，修改 PATH 之前，先运行 MY_OLD_PATH=$PATH，然后修改 PATH 的值。需要将 PATH 的值还原时，运行 PATH=$MY_OLD_PATH 即可。

3.2.3　设置和取消别名命令 alias 和 unalias

假设每天需要进入目录/home/user/var/log/backlogs/project 查看日志，每天需要执行命令 cd /home/user/var/log/backlogs/project。该命令有点长，难记，也容易输入错误。这时可以使用别名功能使其变短。alias 是 Bash 的内置命令，用来定义别名，基本格式为：

```
alias 别名=值
```

例如，定义别名 prjlog，让它等于一条进入固定目录的 cd 命令：

```
$ alias prjlog='cd /home/user/var/log/backlogs/project'
```

定义了别名 prjlog 之后，运行 prjlog 就可以进入目录，无需再输入参数很长的 cd 命令。

运行 alias 或者 alias -p，可以列出（print）当前所有的别名。下面列出某 Linux 系统默认的别名：

```
$ alias -p
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
```

可见别名 l 等于 ls -CF，运行 l 就相当于运行 ls -CF。下面运行 l 试试：

```
$ l
1.q       a.txt        Documents/   kkk/        Public/      sy3.bash*     tmp/
777*      b_ln2.txt    Downloads/   mmm.txt     samba_log@   sy4.txt       Videos/
9999@     b.txt        env2.txt     Music/      set.txt      sy4.txt_ln    z.txt
A.h       Desktop/     examples     Pictures/   sy2.bash*    Templates/
```

顺便解释一下上面的输出。上面以列的形式显示当前目录的内容，尾部为/的表示它是

目录，为*的表示它是可执行文件，为@的表示它是符号链接。运行 man ls，查看 ls 命令的有关选项-C 和-F 的帮助就清楚了。

运行 alias 加上一个别名的名字，可以查看该别名的定义，例如：

```
$ alias l
alias l='ls -CF'
```

运行 alias 加上一个并未定义为别名的名字，将得到该别名未找到的提示，例如：

```
$ alias abc
bash: alias: abc: not found
```

内置命令 unalias 用来删除别名。取消前面已经定义的别名 prjlog，再查看它，发现该别名没有定义了：

```
$ unalias prjlog                    # 取消别名 prjlog
$ alias prjlog                      # 查看别名定义
bash: alias: prjlog: not found      # 该别名定义已不存在
```

命令 unalias -a 用来删除所有的别名定义。

3.2.4　修改.bashrc——设置自己的环境

前面说过，如果希望将命令提示设置为“计算机名#时间>”，运行 PS1="\h#\t>"即可：

```
user@ubuntu:~$ PS1="\h#\t>"
ubuntu#10:18:25>
```

现在，关闭命令行窗口，再打开一个新的 shell 窗口，会发现刚刚设置的 PS1="\h#\t>"怎么不起作用了？本节将回答这个问题。

在自己的主目录下运行 ls -a，可以看见一个名字为.bashrc 的隐藏文件。可以查看一下.bashrc，下面是 Ubuntu 12.04 的.bashrc 的部分内容。

```
$ cat .bashrc
# ~/.bashrc: executed by bash(1) for non-login shells.
......
# If not running interactively, don't do anything
[ -z "$PS1" ] && return
......
# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000
......
# make less more friendly for non-text input files, see lesspipe(1)
[ -x /usr/bin/lesspipe ] && eval "$(SHELL=/bin/sh lesspipe)"
```

```
......
# enable color support of ls and also add handy aliases
if [ -x /usr/bin/dircolors ]; then
	test -r ~/.dircolors && eval "$(dircolors -b ~/.dircolors)" || eval "$(dircolors -b)"
	alias ls='ls --color=auto'
	alias grep='grep --color=auto'
	alias fgrep='fgrep --color=auto'
	alias egrep='egrep --color=auto'
fi

# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'
......
if [ -f ~/.bash_aliases ]; then
	. ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
	. /etc/bash_completion
fi
```

.bashrc 其实是一个脚本，每进入新的 Bash shell 时（具体地应该说：非登录的 shell，见第 10 章），.bashrc 被执行，这就是它的作用。举例说明之前先将.bashrc 的内容复原，如果修改过的话。

用编辑器修改.bashrc，将 PS1="\h#\t>"添加到最后一行，保存，然后，重新打开一个命令行窗口，命令行提示变成了预期的样子。

```
user@ubuntu:~$ gedit .bashrc        # 修改.bashrc
ubuntu#10:28:45>                    # 重新打开一个 shell terminal，系统提示变了
```

重新打开一个命令行窗口一定进入新的 shell。在当前窗口运行命令 bash 也可以进入新的 shell：

```
user@ubuntu:~$ gedit .bashrc        # 或者用 vi 修改.bashrc
user@ubuntu:~$ bash                 # 运行 bash 进入新的 shell
ubuntu#10:30:32>                    # 系统命令行提示变了
```

执行 exit 退出一层 shell，即退回到原来的 shell，命令行提示恢复到原样：

```
ubuntu#10:30:48> exit               # 退出一层 shell
user@ubuntu:~$                      # 提示恢复原样
```

实际上，命令行提示的设置、别名的设置和搜索路径的设置等只是系统环境设置很小的

一部分。对环境的设置，并非只能通过修改.bashrc 文件来实现。以 Ubuntu 系统为例，/etc/environment、/etc/profile 和/etc/bash.bashrc 这三个文件控制着整个系统的环境设置，它们的修改会影响所有用户的环境，而每个用户主目录下的.bashrc 的修改只影响用户自己的环境。

```
$ ls -l /etc/environment /etc/profile /etc/bash.bashrc
-rw-r--r-- 1 root root 1939   2011-03-31 15:26 /etc/bash.bashrc
-rw-r--r-- 1 root root     79   2011-04-29 04:59 /etc/environment
-rw-r--r-- 1 root root   497   2011-04-29 04:59 /etc/profile
```

除了.bashrc 之外，还有.bash_profile，.profile，.bash_login 和.bash_logout 等与账户环境设置相关的文件，见 10.4 节。对于初学者，知道在自己的主目录下有个叫.bashrc 的隐藏文件用于设置自己的环境就基本够了。由于文件.bashrc 的内容直接影响着用户的环境，所以修改它时需要小心谨慎，修改之前最好先备份，即运行 cp .bashrc .bashrc_bak。万一改坏了，可以运行 cp .bashrc_bak .bashrc 恢复它。

3.3　权限掩码命令 umask

创建一个新目录和一个新文件，它们的初始权限是什么？试一试就知道了：

```
$ mkdir test_dir                              # 创建一个新目录（空目录）
$ touch test_file                             # 创建一个新文件（空文件）
$ ls -ld test_dir test_file
drwxr-xr-x  2       user user       4096    2012-10-22 21:48 test_dir
-rw-r--r--    1       user user             0    2012-10-22 21:48 test_file
```

由此可见，目录的权限是 755（rwxr-xr-x），文件的权限是 644（rw-r--r--）。这是它们的默认权限，这个默认值是可以用内置命令 umask 设置的。直接运行 umask 可以显示当前的掩码值：

```
$ umask
0022
```

所谓掩码，简单地说，是一个八进制的三位数，就是从满权限中“拿走”的权限。目录的满权限是 777（rwxrwxrwx），拿走 022 的话，剩下 755（rwxr-xr-x），755 就是掩码为 022 时的新建目录的默认权限。文件在（用 touch 命令）新建时默认没有执行权限，文件的满权限是 666（rw-rw-rw-），若拿走 022，则剩下 644（rw-r--r--），644 就是掩码为 022 时的新建文件的默认权限。

新文件可以来自复制，复制时掩码也是起作用的。先确保文件 a.h 的权限是 666：

```
$ chmod 666 a.h
$ ls -l a.h
-rw-rw-rw- 1 user user 0 2012-10-22 22:10 a.h                # a.h 的权限是 666（rw-rw-rw-）
```

将 a.h 复制到 b.h，新文件 b.h 的权限是 644（rw-r--r--），被拿走了 022：

```
$ cp a.h b.h
$ ls -l a.h b.h
-rw-rw-rw-   1      user user 0 2012-10-22 22:10 a.h
-rw-r--r--   1      user user 0 2012-10-22 22:11 b.h
```

这里顺便讲一个命令 cp 的用于保留权限的选项，它是--preserve=mode 或者-p，使用--preserve=mode 时，新文件的权限将与旧文件一致。例如：

```
$ cp --preserve=mode a.h c.h                              # 此命令也可以是 cp -p a.h c.h
$ ls -l a.h c.h
-rw-rw-rw-   1      user user 0 2012-10-22 22:10 a.h
-rw-rw-rw-   1      user user 0 2012-10-22 22:15 c.h      # c.h 的权限为 666（rw-rw-rw-）
```

使用选项-p 时，新文件的权限及修改时间都与旧文件保持一致。

具体的 umask 的值与文件及目录的权限的对应关系见表 3-3。

umask 命令的选项-p 用来以数字形式显示掩码，选项-S 用来以符号形式显示默认权限，对比一下：

```
$ umask
0022
$ umask -p
umask 0022
$ umask -S
u=rwx,g=rx,o=rx
```

umask 等于 022 时，目录的默认权限为 755（rwxr-xr-x），也就是 u=rwx，g=rx，o=rx。

设置了掩码，也就控制了文件和目录的默认权限。下面将掩码设为 077，看看新建目录和新建文件的默认权限：

```
$ umask 077
$ mkdir tst_dir                # 创建一个新目录（空目录）
$ touch tst_file               # 创建一个新文件（空文件）
$ ls -ld tst_dir tst_file
drwx------  2      user user      4096   2012-10-22 22:33 tst_dir
-rw-------  1      user user         0   2012-10-22 22:33 tst_file
```

可见，掩码设为 077 之后，新建的目录，只有用户自己能访问，并有写权限，其他账户没有任何权限；新建的文件，只有用户自己有读写权限，其他账户没有任何权限。

如果用户已经将某权限掩码值确定下来，并在一段时间内不变的话，可以将 umask 命令放入自己主目录下面的.bashrc 文件中。比较常用的 umask 值见表 3-4。

本小节前面对掩码的解释并不完全透彻，在复制文件时，掩码的作用是拿走其能够拿走的权限。假设文件 x.h 的权限是 756，掩码仍为 022，运行 cp x.h y.h 之后，y.h 的权限是 644，对吗？从 756 中拿走 022，即，从 7=4+2+1 中拿走 0，什么也没拿；从 5=4+1 中拿走 2，4+1 中没有 2，什么也没拿；从 6=4+2 中拿走 2，剩余 4，则 y.h 的权限为 754（rwxr-xr--）。看实例：

表 3-3 umask 值与权限对应关系

umask 值	目录权限	文件权限
0	7	6
1	6	6
2	5	4
3	4	4
4	3	2
5	2	2
6	1	0
7	0	0

表 3-4 常用的 umask 值与相应的权限

umask 值	目录权限	文件权限
022	755	644
027	750	640
002	775	664
006	771	660
007	770	660

```
$ touch x.h; chmod 756 x.h
$ ls -l x.h
-rwxr-xrw- 1 user user 0 Feb 12 18:36 x.h
$ umask 022
$ cp x.h y.h
$ ls -l y.h
-rwxr-xr--   1 user user 0 Feb 12 18:38 y.h                # 可见 y.h 的权限与前面的分析一致
```

3.4 source 命令和点命令

根据前面的讲解，修改了.bashrc 之后，似乎必须重新打开一个命令行窗口（或运行 bash），进入一个新的 shell terminal，修改才能起作用。其实不然，使用内置的 source 命令或者“.”命令（也叫点命令），可以使.bashrc 的修改马上生效。举例之前先将前面的.bashrc 的内容复原，将命令行提示恢复到原来的值。

```
user@ubuntu:~$          # PS1 为“账户名@计算机名:路径$”
```

用 vi 或者 gedit 编辑.bashrc，将 PS1="\h#\t>"添加为.bashrc 的最后一行：

```
user@ubuntu:~$ gedit .bashrc
```

运行 source .bashrc 或. .bashrc（而不是进入一个新的 shell terminal），.bashrc 的修改马上生效，说明文件.bashrc 中的命令得到了执行：

```
user@ubuntu:~$ source .bashrc
ubuntu#07:26:19>
```

“.”命令和 source 命令的格式如下，即“.”和 source 的后面跟文件名（一般就是脚本名）和参数（如果脚本有参数）：

```
. 文件名 [参数]
source 文件名 [参数]
```

例如，z.txt 文件里面有三条命令：

```
$ cat z.txt
echo "hello z.txt"              #  显示 hello z.txt
date                            #  显示当前时间
hostname                        #  显示计算机名
```

运行 source z.txt，那么文件 z.txt 中的三条命令依次执行：

```
$ source z.txt
hello z.txt
Sun Sep  9 06:01:08 CST 2012
ubuntu
```

运行命令 source file 的时候，如果 file 不在当前目录下，但在搜索路径$PATH 中可以找到文件 file，命令 source file 也可以正常运行。

3.5　编写并运行第一个脚本

假设已经编辑好了一个包含三行内容的脚本 first.sh：

```
$ cat first.sh
#!/bin/bash
# This is the first bash script written by me
echo "I like bash."
```

第一行中的/bin/bash 为此脚本的解释程序；第二行以#开头，是注释行，如果#位于行首，则整行均为注释，如果#不位于行首，那么#的后面为注释；第三行为一条可执行的 Linux 命令。运行脚本：

```
$ first.sh
first.sh: command not found
```

当前目录下面明明有这个文件，为什么执行它的时候，会遇到“命令没有找到”的提示呢？查看一下搜索路径和当前路径：

```
$ echo $PATH
/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
$ pwd
/home/user
```

可见，当前路径不在搜索路径里面。如果当前路径不在搜索路径里面时，有两种解决方法。

第一种方法，可以在脚本名字前面加./，./就是指明它在当前目录下，即运行./first.sh 就不会遇到提示 command not found 了：

```
$ ./first.sh
bash: ./first.sh: Permission denied
```

command not found 的提示没有了，可又遇到“没有执行权限（Permission denied）”的提示。马上会讲如何解决。

第二种方法，当前路径不在搜索路径里面时，可以把当前路径（即“.”）加入到搜索路径里面，用冒号间隔（将下面这句命令加到文件.bashrc 里面，就不用每次执行它了）：

```
$ PATH="$PATH:."
```

再查看环境变量 PATH，PATH 的尾部增加了当前路径（即“.”）：

```
$ echo $PATH
/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:.
```

现在运行脚本：

```
$ first.sh
bash: ./first.sh: Permission denied
```

与第一种方法的情形一样。用 chmod 命令增加它的执行权限，再运行，就能得到预期的结果：

```
$ chmod +x first.sh
$ first.sh
I like bash.
```

当脚本没有执行权限或者当前路径不在搜索路径里时，可以使用前面讲过的点命令或者 source 命令，即运行 source <脚本名>：

```
$ source first.sh
I like bash.
```

运行 source 加脚本名时，即使脚本有没有执行权限、当前路径在不在搜索路径里，脚本也可以正常运行。但是用 source <脚本名>来运行脚本的方法使用得相对少一些，直接运行可执行脚本使用得较多。接下来还有很多运行脚本的例子，在后面的章节中，默认搜索路径已经包含了当前路径，并且默认脚本都有执行权限（默认已用 chmod +x 增加了执行权限）。

书上的脚本，在自己的 Linux 系统中有时候无法正常运行。这时候，首先需要检查自己的 Bash 版本是否太老（虽然计算机和服务器更新得越来越快，但这种可能性还是有的），即运行命令 bash --version。然后，需要检查自己的 Bash 工作环境。关于环境，在接下来的章节里有进一步的阐述。

3.6 命令解释顺序与命令类型的查询

3.6.1 命令的解释顺序

内置命令、关键字、外部命令（可执行程序和脚本，包括系统提供的，还包括用户自己写的、复制的、安装的程序和脚本）、别名，还有后面要讲到的函数，都是可以被调用执行

的，可以相互重名（但是最好避免重名）。例如，当设置一个别名时，可以让这个别名与某个内置命令同名，那么，运行这个重名的命令时，是别名被执行了呢，还是内置命令被执行了呢？这需要知道 Bash 的命令解释顺序：

```
alias→keyword→function→built-in→$PATH
```

即，一旦出现重名的命令，那么优先级由高到低，依次是：别名→关键字→函数→内置命令→外部命令。也可以这样理解，当运行一条命令时，系统依次在别名列表、关键字列表、函数列表和内置命令列表里面查找，一旦找到就立刻执行它并停止后续的查找；如果还没有找到，则会按照环境变量 PATH 里面存放的搜索路径的先后顺序，在外部命令中继续查找，一旦找到就立刻执行它并停止在后边路径里面的查找。如果最终还是没有找到，系统将提示 command not found。

3.1 节提到，内置 echo 命令优先于外部的同名命令。现在，可以很容易地理解了。

3.6.2　改变命令解释顺序的三条内置命令

Bash 处理命令的顺序并非不可改变。Bash 提供的三条内置命令可以改变命令的处理顺序。

1）command <命令>，忽略别名和函数，按照内置命令和外部命令来处理。

2）builtin <命令>，只查找内置命令，忽略函数和外部命令。

3）enable，禁用或使能内置命令。

下面分别举例说明。

pwd 的作用原本是显示当前目录，下面设置一个名字为 pwd 的别名：

```
$ alias pwd='echo this is alias of pwd'
```

因为别名的优先级最高，所以执行 pwd 就会执行别名 pwd，而不是原本的 pwd 命令：

```
$ pwd
this is alias of pwd          # 执行的是刚定义的别名 pwd，而不是原本的 pwd 命令
```

这时，要想执行原本的 pwd，前面需要加上 command。command pwd 的作用是，不在别名列表和函数列表里面查找 pwd，而是在内置命令和外部命令里面查找并运行 pwd，这样，原本的内置命令 pwd 得到执行：

```
$ command pwd
/home/user                    # 原本的 pwd 命令执行了，显示当前目录
```

builtin 的作用是只查找内置命令，否则会给出“不是内置命令（not a shell builtin）”的提示。下面这条命令的意思是运行内置命令 pwd：

```
$ builtin pwd
/home/user
```

因为 ls 不是内置命令，所以 builtin 后面的 ls 命令不会被执行。

```
$ builtin ls -l
```

```
bash: builtin: ls: not a shell builtin          # 显示：ls 不是 shell 内置命令
```

enable 的选项-n 用来禁用内置命令。例如，禁用内置命令 alias 后，将无法使用它：

```
$ enable -n alias                    # 禁用内置命令 alias
$ alias -p                           # 显示当前别名列表
alias: command not found             # 内置命令 alias 现在不可用了
```

原本 alias -p 命令可以显示当前的别名列表，禁用 alias 后，遇到了 command not found 的提示。

再运行下面的命令，使能内置命令 alias，然后内置命令 alias 就可以正常使用了。

```
$ enable alias                       # 使能内置命令 alias
```

因为外部程序的优先级不如内置命令高，当自己写的脚本与某个内置命令重名、并希望默认情况下自己的脚本而不是内置命令被执行时，可以用 enable -n 命令来禁用相应的内置命令。但是，正确的做法是，尽可能不要让自己写的脚本和函数与已有的命令（包括内置的与外部的）重名。

接本章第 1 节继续讨论。操作系统也提供了内置命令，以 Ubuntu 为例，下面的这几条命令，内置命令包含它们，外部命令当中也有它们。

```
$ which pwd echo printf
/bin/pwd
/bin/echo
/usr/bin/printf
```

重名情况下，可以在前面加 builtin 来确保内置命令被执行，或者用命令 enable -n 来禁止某条内置命令被执行，从而使外部命令得到执行的机会。

3.6.3　命令类型的查询

执行一条命令之前，如果想知道它是什么，例如，想知道它是别名还是内置命令，如何操作呢？可以使用内置命令 type，带上选项-t 查询。查询结果可能会是 alias、keyword、function、builtin 和 file，分别表示别名、关键字、函数、内置命令和外部命令。查询结果为空时，表示未找到该命令。下面举例。

```
$ type -t while
keyword          # while 是关键字
$ type -t touch
file             # touch 是外部命令
$ type -t /bin/pwd
file             # 带全路径的 pwd 命令是外部命令
$ type -t pwd
builtin          # 由于内置命令优先，不带路径的 pwd 命令是内置命令
```

定义一个别名：

```
$ alias prjlog='cd /home/user/var/log/backlogs/project'
```

```
$ type -t prjlog
alias                  # prjlog 是别名
```

不带-t 选项的话，显示的信息会略有不同。接前面的例子，做个对比：

```
$ type while
while is a shell keyword          # while 是关键字
$ type touch
touch is /usr/bin/touch           # touch 是外部命令/usr/bin/touch
$ type alias
alias is a shell builtin          # alias 是内置命令
$ type prjlog                     # prjlog 是别名，相当于 cd …命令
prjlog is aliased to `cd /home/user/var/log/backlogs/project'
```

3.7　命令的退出状态以及命令 true 和 false

在一个命令或程序结束之后，都会返回一个退出状态。状态值范围为 0 到 255，0 表示成功执行，非零均表示失败，其中 127 表示未找到命令，内置变量$?存储着上一条命令的退出状态。

例如，date 命令运行成功，退出状态为 0：

```
$ date
Fri Oct 12 00:08:35 CST 2012
$ echo $?
0
```

dateee 命令不存在，找不到该命令，退出状态为 127：

```
$ dateee
dateee: command not found
$ echo $?
127
```

cp 命令的源文件 xy.txt 不存在，复制失败，退出状态为 1：

```
$ cp xy.txt ab.txt
cp: cannot stat `xy.txt': No such file or directory
$ echo $?
1
```

Bash 有两个特殊的内置命令：true 和 false。命令 true 的返回结果永远是成功，命令 false 的返回结果永远是失败。它们的功能类似于其他编程语言的布尔类型。

```
$ true             # 运行命令 true
$ echo $?
0                  # 退出状态为 0，命令 true 总是成功的
$ false            # 运行命令 false
$ echo $?
1                  # 退出状态为 1，命令 false 总是失败的
```

3.8　管道

日常生活中的管道主要用来输送自来水，一头是水的输入，另一头是输出。Linux 系统也有类似于生活中的管道的概念，键盘上的竖线“|”就是管道线。查看一个长文件的前 10 行除了用命令 head <文件名>，还有一种方法，就是命令 cat <文件名> | head，例如：

```
$ cat /etc/passwd | head
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
```

那么如何理解 cat <文件名> | head 这条命令呢？cat <文件名>在管道线的左边，cat 命令输出的内容通过管道被输送给 head 命令，head 命令收到后，列出收到的内容的前 10 行，动作完成，如图 3-1 所示：

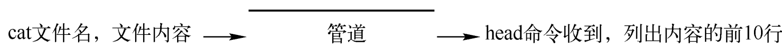


图 3-1　cat 命令、管道与 head 命令

查看前 5 行，运行命令 cat <文件名> | head -5。

同样，分页浏览长文件除了可以用 more <文件名>之外，还可以用 cat <文件名> | more；统计文件除了可以用 wc <文件名>之外，还可以用 cat <文件名> | wc。第 2 章讲过的 email.txt 文件，可以利用管道统计它的字数：

```
$ cat email.txt
Hello Jack,
I'm in China.
Mike
$ cat email.txt | wc
        3        6        31
```

cat 命令输出的 email.txt 的内容通过管道被输送给 wc 命令，wc 命令收到后，列出了内容的行数、单词数和字节数。

ls -l 命令列出当前目录下面的文件和子目录的信息。如果当前目录下的文件和子目录很多，运行 ls -l 命令后，用户只能看见后面的文件和子目录（默认按照文件名的字典顺序排

列），前面的内容已经“呼啸而过”。想查看全部内容，当然可以用鼠标滚动屏幕，但还有一种方法，就是使用管道加命令 more。

运行 ls -l | more 就可以分页查看 ls 命令的输出内容。运行 ls -l | head 可以查看按字典顺序排在前面的文件。运行 ls -lt | head 就可以查看修改时间最新的文件。

要想知道当前目录下面的文件和子目录的个数可以使用 ls 命令，数一数文件和子目录的个数。还有快一些的方法，就是使用 ls 命令、管道和 wc 命令：

```
$ ls | wc -w
40
$ ls -a | wc -w
70
```

说明当前目录下面的非隐藏的文件和子目录是 40 个，所有的（包括隐藏的）文件和子目录是 70 个，可知，隐藏的文件和子目录是 30 个。

3.9　输入输出重定向

3.9.1　标准输入和标准输出

执行一个 shell 命令会自动打开标准输入文件（stdin）和标准输出文件（stdout），它们默认分别对应终端键盘和终端屏幕，分别对应文件描述符 0 和文件描述符 1。

运行 ls -l /dev/std*命令（其中 stderr，后面马上会讲）：

```
$ ls -l /dev/std*
lrwxrwxrwx 1 root root 15 Jun 23 03:05 /dev/stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root 15 Jun 23 03:05 /dev/stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root 15 Jun 23 03:05 /dev/stdout -> /proc/self/fd/1
```

文件描述符（file descriptor，简写为 FD 或 fd），是进程对其所打开文件的索引，形式上是个非负整数。

前面讲过命令 wc <文件>，文件是命令 wc 的输入，运行的结果，即文件的行数、单词数和字节数是命令 wc 的输出。下面以 wc 命令为例，感受一下标准输入，即让 wc 命令的输入来自终端键盘而不是来自文件。输入 wc，按回车键，发现屏幕不动了，其实并未死机，这时 shell 在等着用户的键盘输入，输入如下的三行内容，再按〈Ctrl+D〉键结束：

```
$ wc
Hello Mike,
See you tomorrow, have a nice sleep.
Tom
        3        10        53
```

按〈Ctrl+D〉键后，wc 命令对刚刚输入的内容的统计结果就马上出来了。

下面以 cat 命令为例，感受一下标准输入和标准输出，即让输入来自终端键盘，输出到屏幕终端。输入 cat，按回车键，任意输入一些句子，按回车键，可以发现，刚刚输入的内

容马上显示在屏幕上：

```
$ cat
I study bash                    # 用户的键盘输入
I study bash                    # 命令 cat 的输出
this is cat command             # 用户的键盘输入
this is cat command             # 命令 cat 的输出
I input one line                # 用户的键盘输入
I input one line                # 命令 cat 的输出
Bye                             # 用户的键盘输入
Bye                             # cat 命令的输出
```

按〈Ctrl+C〉键或〈Ctrl+D〉键可以结束 cat 命令的运行。可见，Linux 下的典型的命令的执行过程，如图 3-2 所示：

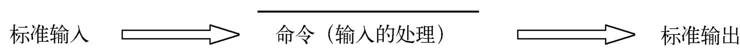


图 3-2　输入来自终端键盘，输出到终端屏幕的命令的执行过程

3.9.2　输入输出重定向与输出追加重定向

使用键盘输入有一个明显的缺点，就是每次都需要用户手工输入，输错的话还要重来，效率不高。其实，输入是可以重定向的，输入可以不来自键盘，而来自文件。输入重定向的格式为：

```
命令 < 文件名
```

取 cat 和 wc 命令的输入重定向文件为前面提到过的 email.txt：

```
$ cat < email.txt
Hello Jack,
I'm in China.
Mike
$ wc < email.txt
3   6   31
```

不难看出，cat < email.txt 和 cat email.txt 效果不是一样的吗？cat < email.txt 表示 cat 命令要显示的内容来自文件 email.txt（而不是来自键盘的输入），cat email.txt 表示要显示文件 email.txt 的内容，效果是一样的。既然一样，那么介绍输入重定向有必要吗？有必要，后面讲 while 循环的时候将会再举输入重定向的例子，那时对输入重定向会有更深刻的认识。

命令行的输出默认显示在终端屏幕上，这有时会给用户带来不便。一旦终端 shell 窗口关闭，或者屏幕经过了长长的滚动，再想查看刚才某条命令行的输出，就不容易了，甚至是不可能了。输出也可以重定向到文件，格式为：

```
命令 > 文件名
```

查看文件 email.txt 的信息，运行 ls 命令，信息输出在屏幕上：

```
$ ls -l email.txt
-rw-rw-r-- 1 user user 31 2012-10-25 11:23 email.txt
```

如果将 ls 命令的输出重定向到了文件 lst.txt，屏幕上将不显示文件 email.txt 的信息：

```
$ ls -l email.txt > lst.txt
$                        # 屏幕无输出信息
```

ls -l email.txt 命令输出了什么？终端屏幕上没有显示，因为 ls -l email.txt 显示的信息重定向到了文件 lst.txt 里，查看一下 lst.txt 就知道了：

```
$ cat lst.txt
-rw-rw-r-- 1 user user 31 2012-10-25 11:23 email.txt
```

下面将 date 命令的输出先存入文件 time.txt，再查看一下 time.txt 的内容：

```
$ date > time.txt
$ cat time.txt
Thu Oct 25 11:37:18 CST 2012
```

只要 time.txt 文件不被删除，那么运行 date 命令那一刻的时间就被记录了下来。有时，某些脚本需要运行较长时间又需要计时。此时可以在脚本的第一句写命令 date > start_time；在脚本的逻辑上的最末一句，或者说在脚本结束的地方，写命令 date > end_time；根据文件 start_time 和 end_time 中记录的时间，可以计算出脚本的总运行时间。

输入输出重定向通常简称为 I/O 重定向，I/O 是 Input（输入）/Output（输出）的意思。

输出还可以追加重定向，输出追加重定向的格式为（注意两个大于号之间不能有空格）：

```
命令 >> 文件名
```

接着前面的例子，再次运行 date，将此刻 date 命令的输出追加到 time.txt 文件：

```
$ date >> time.txt
$ cat time.txt
Thu Oct 25 11:37:18 CST 2012
Thu Oct 25 11:46:14 CST 2012
```

第二次运行 date 时的时间被追加存到了 time.txt 中。这时，如果再将 date 的输出重定向（而不是追加重定向）到 time.txt，看看会怎样：

```
$ date > time.txt
$ cat time.txt
Thu Oct 25 11:51:49 CST 2012
```

因为不是追加重定向，time.txt 中的旧内容被冲掉了。运行 date > time.txt 时，新的时间存入 time.txt 文件，覆盖了原有的内容。所以在输出重定向到某个文件之前，最好先检查文件是否已经存在（在命令行，可以用命令 ls 检查；学习完第 5 章的判断命令后，就可以知道在脚本里如何检查了）。

下面使用 cat 命令将键盘输入重定向到文件 file1.txt，按〈Ctrl+D〉键结束键盘输入：

```
$ cat > file1.txt
This is file_1                  # 键盘输入，然后回车，再按〈Ctrl+D〉键
```

再将键盘输入重定向到文件 file2.txt，按〈Ctrl+D〉键结束键盘输入：

```
$ cat > file2.txt
This is file_2                  # 键盘输入，然后回车，再按〈Ctrl+D〉键
```

文件 file1.txt 和 file2.txt 就产生了。可以将 file1.txt 与 file2.txt 合并为一个新文件：

```
$ cat file1.txt file2.txt > file.txt
```

cat 命令的参数 file1.txt 在 file2.txt 的前面，所以 file1.txt 的内容在前，file2.txt 的内容在后，组成了新文件 file.txt。查看一下新文件的内容：

```
$ cat file.txt
This is file_1
This is file_2
```

也可以将 file1.txt 的内容追加到 file2.txt 的后面，那么 file2.txt 的内容就更新了，file2.txt 的前半部分内容是它原来的，后半部分内容来自 file1.txt：

```
$ cat file1.txt >> file2.txt
$ cat file2.txt
This is file_2
This is file_1
```

cat 可以将更多的文件合并为一个新文件，如，cat file1 file2 file3 > file，三个文件的内容按照顺序连接在一起组成新文件 file。也可以将更多的文件按照顺序连在一起并追加到某个文件的尾部，如，cat file1 file2 file3 file4 file5 >> file。现在知道 cat 命令的名字是怎么来的了，它是 catenate 或 concatenate（连接）的缩写。

Bash 还识别一种特殊的输出重定向：大于号的左边没有命令（空操作），右边跟一个文件名。即：

```
> 文件名
```

例如：

```
$ > file.txt                          # 将“空操作”重定向到文件
```

文件 file.txt 不存在时，命令“> file.txt”将产生一个新的空文件 file.txt；如果 file.txt 事先已经存在，file.txt 原有的内容将消失，字节数变为 0。除了 touch 命令之外，这里又介绍了一种产生新的空文件的方法。注意，file.txt 事先不存在的话，命令> file.txt 与命令 touch file.txt 效果相同；file.txt 事先存在的话，命令 touch file.txt 只改变了文件的时间戳，文件内容不受影响，命令> file.txt 将文件的内容清空（文件字节数变为 0），当然也同时改变文件的时间戳。

输入重定向与输出重定向可以在一条命令里同时使用。例如，将文件 email.txt 复制到 email_bak.txt，当然会首先想到命令：cp email.txt email_bak.txt。其实还可以用下面的命令：

```
$ cat < email.txt > email_bak.txt
```

这条命令的意思是，将文件 email.txt 作为命令 cat 的输入，命令 cat 显示出来的 email.txt 的内容被重定向到 email_bak.txt（而不是终端屏幕），这样的话，email_bak.txt 的内容与 email.txt 完全一致：

```
$ cat email_bak.txt
Hello Jack,
I'm in China.
Mike
```

复制文件用命令 cp file1 file2，cat file1 > file2 和 cat < file1 > file2 都是可以的。一定不要认为 cat < file1 > file2 中的两个重定向符号是一对普通的尖括号。

3.9.3　预防输出重定向覆盖旧文件

从前面的内容似乎可以得出结论：某条命令输出重定向到某个事先已存在的文件的话，该文件将被覆盖，原有的内容将丢失。例如：

```
$ cat a.txt
I am a.txt                          #  查看 a.txt 的内容
$ echo "I am not b.txt" > a.txt     # echo 命令输出重定向到 a.txt
$ cat a.txt
I am not b.txt                      # a.txt 的内容变了
```

通过设置 Bash 环境中的选项 noclobber，可以控制重定向时是否覆盖已存在的文件。Bash 环境中的选项 noclobber 默认是关闭的，所以，重定向时默认会覆盖已存在的文件。运行 set -o 命令可以查询选项是打开的还是关闭的。运行 set -o 时，全部的 set 命令可以控制的选项都列出来了，每个选项的打开与关闭控制着相应的命令的行为方式。本节只介绍 noclobber。

```
$ set -o
allexport               off
braceexpand             on
emacs                   on
errexit                 off
errtrace                off
functrace               off
hashall                 on
histexpand              on
history                 on
ignoreeof               off
interactive-comments    on
keyword                 off
```

```
monitor         on
noclobber       off             # 选项 noclobber 默认是关闭的（off）
noexec          off
noglob          off
nolog           off
notify          off
nounset         off
onecmd          off
physical        off
pipefail        off
posix           off
privileged      off
verbose         off
vi              off
xtrace          off
```

可以使用管道线加 grep noclobber，只列出 noclobber 的状态：

```
$ set -o | grep noclobber
noclobber       off
```

要打开该选项，可以用命令 set -C，或者用如下命令：

```
$  set -o noclobber                 # 打开选项 noclobber
$  set -o | grep noclobber
noclobber       on                  # 查看一下，可见选项 noclobber 打开了（on）
```

这时，输出重定向到 a.txt 将会失败并得到提示：

```
$ echo "I am not c.txt" > a.txt
bash: a.txt: cannot overwrite existing file       # 不能覆盖已存在的文件
```

要关闭选项 noclobber，可以用命令 set +C，或者用如下命令：

```
$ set +o noclobber                  # 关闭选项 noclobber
$ set -o | grep noclobber
noclobber       off                 # 查看一下，可见选项 noclobber 关闭了
```

所以，在输出重定向时，如果不希望出现旧文件被覆盖的情况，要么先检查文件是否已经存在，要么将选项 noclobber 打开。

3.9.4 标准错误输出与“黑洞”

执行一个 shell 命令还会自动打开标准错误输出文件（stderr），它默认也是对应终端屏幕，对应文件描述符 2。下面用 ls 命令查看一个不存在的文件的信息：

```
$ ls -l email_bak
ls: cannot access email_bak: No such file or directory
```

No such file or directory 的信息输出在屏幕上，但这不再是标准输出，而是标准错误输

出。如何理解标准错误输出与标准输出的区别呢？下面试图将信息 No such file or directory 重定向到文件 lst_bak.txt 里面，结果发现，No such file or directory 仍然显示在屏幕上，而文件 lst_bak.txt 是个空文件：

```
$ ls -l email_bak > lst_bak.txt
ls: cannot access email_bak: No such file or directory
$ cat lst_bak.txt
$                               # 无任何显示，说明 lst_bak.txt 是个空文件
```

需要使用 2>对标准错误输出进行重定向，使用 2>>对标准错误输出进行追加重定向，注意，2 与>之间不能有空格。把前面例子中的>变为 2>，No such file or directory 不再显示在屏幕上，而是保存到了文件 lst_bak.txt 里面：

```
$ ls -l email_bak 2> lst_bak.txt
$ cat lst_bak.txt
ls: cannot access email_bak: No such file or directory
```

一条命令的结果，常常既有正常输出信息，又有错误信息：

```
$ ls -l email.txt email_bak
ls: cannot access email_bak: No such file or directory        # 错误提示，文件不存在
-rw-rw-r-- 1 user user 31 2012-10-25 11:23 email.txt          # 正常输出信息
```

可以将这两类信息分别重定向到不同的文件（下面命令中的 1>可以简写为>），标准输出重定向到 lst.txt，标准错误输出重定向到 lst_bak.txt：

```
$ ls -l email.txt email_bak 1> lst.txt 2> lst_bak.txt
$ cat lst.txt
-rw-rw-r-- 1 user user 31 2012-10-25 11:23 email.txt          # 正常输出信息
$ cat lst_bak.txt
ls: cannot access email_bak: No such file or directory        # 错误提示信息
```

也可以重定向到同一个文件。在下面的命令中，> lst.txt 的作用是将标准输出重定向到文件 lst.txt，2>&1 的作用是将标准错误输出重定向到标准输出（可以理解为：标准错误输出“跟随着”标准输出，标准输出重定向到哪里，标准错误输出就重定向到哪里）。那么下面命令的正常输出信息和错误信息都重定向到了文件 lst.txt 中：

```
$ ls -l email.txt email_bak > lst.txt 2>&1
$ cat lst.txt
ls: cannot access email_bak: No such file or directory        # 错误提示信息
-rw-rw-r-- 1 user user 31 2012-10-25 11:23 email.txt          # 正常输出信息
```

此外，使用&>或>&，也可以将标准错误输出和标准输出重定向到同一个文件：

```
$ ls -l email.txt email_bak &> lst.txt
$ cat lst.txt
ls: cannot access email_bak: No such file or directory
-rw-rw-r-- 1 user user 31 2012-10-25 11:23 email.txt
```

标准错误输出与标准输出，都是输出，默认都显示在屏幕上，为什么要把它们分开呢？因为日常工作中的 Linux 服务器，经常有很多程序在运转着，还有维护人员定时地查看日志，将正常信息和异常信息分开，有利于及时发现问题、定位问题。

某些情况下，一个脚本的运行，只需要知道其正常输出信息，错误信息可以忽略。此时可以将错误信息重定向到设备文件/dev/null。/dev/null 是空设备，是 Linux 的一个特殊文件，它是“垃圾桶”“无底洞”，或者叫它“黑洞”，任何输出重定向到它的话，就消失得一干二净。

例如，查看一个不存在的文件，会遇到 No such file or directory 的提示：

```
$ cat email_bak
cat: email_bak: No such file or directory
```

如果不希望看到错误提示，可以将标准错误输出重定向到“黑洞”：

```
$ cat email_bak 2> /dev/null
$                          # 不再有错误提示
```

某些时候，有的命令或者脚本的正常输出信息也不需要关心，可以将其重定向到“黑洞”，即，标准输出也可以重定向到“黑洞”。

```
$ echo "Today is Thursday"
Today is Thursday
```

运行下面的命令，则屏幕上没有输出：

```
$ echo "Today is Thursday" 1> /dev/null          # 命令中的“1>”可以简写为“>”
$                                                # 屏幕没有输出
```

Bash4 版本新增了两个重定向符号：&>>和|&。command &>> file 等价于 command >> file 2>&1，意思是，命令 command 的标准错误输出和标准输出一起追加重定向到文件 file。在下一小节介绍|&的使用。

3.9.5　同时把结果输出至标准输出与文件的命令 tee

单词 tee 是字母 T 的意思，从外形看，T 显然是一个“三通”。命令的结果默认显示在屏幕上，若命令的结果被重定向到一个文件，那么屏幕上就看不到了。如果希望命令的结果显示在屏幕上的同时，又能被重定向到一个文件，可使用三通命令 tee。例如，执行命令 ls -l 123.log，文件 123.log 的信息显示在屏幕上，后面跟着管道线和 tee loginfo.txt 时，屏幕上显示的内容同时也存入了 loginfo.txt：

```
$ ls -l 123.log | tee loginfo.txt
-rw-r--r-- 1 user user 29 2013-03-04 12:32 123.log
$ cat loginfo.txt
-rw-r--r-- 1 user user 29 2013-03-04 12:32 123.log
```

可以用一张图来示意上面使用管道和命令 tee 的例子，如图 3-3 所示。

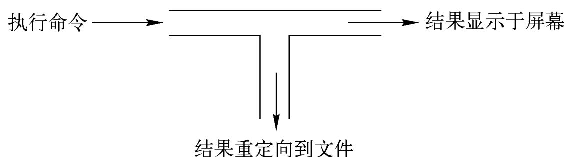


图 3-3　<命令> | tee <文件>示意图

例如，查找当前目录下名字以.txt 结尾的文件，并将查找结果存入 txt_list：

```
$ find -name "*.txt" | tee txt_list
./colleague.txt
./email.txt
./tmp/cronjob.txt
./exec_test.txt
```

查看一下 txt_list，看看是否与上面命令的屏幕显示结果一致：

```
$ cat txt_list
./colleague.txt
./email.txt
./tmp/cronjob.txt
./exec_test.txt
```

使用命令 tee file 时，如果 file 是个已经存在的文件，那么 file 的老内容会丢失，被新内容冲掉。例如：

```
$ date | tee txt_list
Fri Nov 29 16:28:06 CST 2013
$ cat txt_list
Fri Nov 29 16:28:06 CST 2013
```

使用 tee 命令的选项-a 或--append 时，即，使用命令 tee -a file 时，新内容会附加在文件 file 的后面，file 的老内容不会丢失。例如：

```
$ echo "Now I append one line" | tee -a txt_list
Now I append one line
$ cat txt_list
Fri Nov 29 16:28:06 CST 2013
Now I append one line
```

如果 tee 命令后面的文件名用“-”代替，内容会显示在屏幕上（而不是重定向到文件），这时，相同的内容会显示两次，例如：

```
$ ls -l 123.log | tee -
-rw-r--r-- 1 user user 29 2013-03-04 12:32 123.log
-rw-r--r-- 1 user user 29 2013-03-04 12:32 123.log
```

下面命令的作用是，从根目录开始查找名字以.txt 结尾的文件，在屏幕上将显示正常信息（找到的名字以.txt 结尾的文件）和错误信息（如 Permission denied，有的目录用户没有权

限进入搜索），找到的文件列表存入 txt_list：

```
$ find / -name "*.txt" | tee txt_list
```

如果希望正常信息和错误信息都存入 txt_list，用如下命令：

```
$ find / -name "*.txt" 2>&1 | tee txt_list
```

Bash4 新增了重定向符号|&，command1 |& command2 等价于 command1 2>&1 | command2。所以上面的命令可以变为：

```
$ find / -name "*.txt" |& tee txt_list
```

总结前面的内容，得到表 3-5。

表 3-5　常用的重定向命令

命令格式	含　义
command > file	标准输出重定向到文件（而不是终端屏幕）
command 1> file	同上
command >> file	标准输出追加重定向到文件
command 2> file	标准错误重定向到文件
command 2>> file	标准错误追加重定向到文件
command > file 2> file2	标准输出重定向到文件 file，标准错误重定向到文件 file2
command > file 2>&1	标准输出与标准错误一起重定向到文件
command &> file	同上
command >& file	同上
command >> file 2>&1	标准输出与标准错误一起追加重定向到文件
command &>> file	同上，Bash4 支持
command < file	命令以文件（而不是终端键盘）作为标准输入
command < file > file2	以 file 作为命令的输入，标准输出重定向到 file2
command | tee file	命令的结果显示在终端屏幕，结果同时存入 file
command |& command2	command 的标准输出与标准错误一起通过管道传给 command2，Bash4 支持，等价于 command 2>&1 | command2

文件描述符 0，1，2 分别对应标准输入、标准输出、标准错误。其他的数字也能作为文件描述符进行重定向。下一节先介绍 exec 命令，然后继续文件描述符的讲解。初学者可以暂时跳过下一节。

3.10　exec 命令与文件描述符

Bash 的内置命令 exec 的常用格式如下：

```
exec [command [arguments ...]]
```

即，exec 加命令及其选项和参数。exec 不启动新的 shell，而用被执行的命令替换当前的 shell 进程，并且将老进程的环境清理掉，exec 命令后的其他命令将不再执行。例如，在一个 shell 窗口里面运行 exec date，将显示当前的日期和时间，然后，shell 窗口马上就退出或关闭了（几乎还没看清 date 命令的显示结果，就退出了）。这是因为 shell 窗口的进程已经被替换为仅仅执行 date 命令的一个进程，date 运行结束，shell 自然就退出了。所以，很少在命令行把一条普通命令放在 exec 的后面。下面试着在脚本里使用 exec。先看一个脚本。

```
$ cat echo_test.sh
#!/bin/bash
echo "Good morning"
echo "Good afternoon"
```

脚本 echo_test.sh 只有两条 echo 命令，运行结果如下：

```
$ echo_test.sh
Good morning
Good afternoon
```

如果把两条 echo 前面都加上 exec，得到脚本 exec_echo_test.sh：

```
$ cat exec_echo_test.sh
#!/bin/bash
exec echo "Good morning"
exec echo "Good afternoon"
```

运行它，看看会怎样：

```
$ exec_echo_test.sh
Good morning
```

只有第一条命令运行了。因为，当第一条命令 exec echo 运行后，脚本 exec_echo_test.sh 的子 shell 退出了，所以第二条 exec echo 命令不运行。通过这个例子可知，在脚本里面，也不能随意使用命令 exec。

当 exec 命令用来对文件描述符操作（用来重定向）的时候，像其他普通命令一样，不会替换当前 shell，格式如下：

```
exec [redirection ...]
```

exec 与重定向及文件描述符相关的命令见表 3-6。

表 3-6　exec 命令与文件描述符

命　令	作　用
exec command	执行 command，执行完后不返回原来的 shell
exec < file	将 file 的内容作为 exec 的标准输入
exec > file	标准输出重定向到文件
exec m< file	将 file 读入到文件描述符 m 中
command <& m	文件描述符 m 作为命令的标准输入
exec m> file	将写入文件描述符 m 的内容重定向到文件 file 中
command >& m	命令的标准输出重定向到文件描述符 m
exec n<& m	创建文件描述符 m 的副本——文件描述符 n
exec m<& -	关闭文件描述符 m
exec <& -	关闭标准输入
exec >& -	关闭标准输出

下面讲解 exec < file 的使用。先看一个包含三行命令的文件

exec_test.txt：

```
$ cat exec_test.txt
pwd
date
echo "I study bash now."
```

可以用 source 命令或者点命令运行它：

```
$ source exec_test.txt                # 运行. exec_test.txt 也可以
/home/user                            # pwd 的运行结果
Tue Feb 26 06:10:43 EST 2013          # date 的运行结果
I study bash now.                     # echo 的运行结果
```

运行 exec < exec_test.txt，也能得到三条命令的运行结果，但还没来得及看清楚，shell 窗口就关闭了：

```
$ exec < exec_test.txt
/home/user                            # pwd 的运行结果
Tue Feb 26 06:12:28 EST 2013          # date 的运行结果
I study bash now.                     # echo 的运行结果
$ exit                                # 自动运行 exit，退出 shell
```

下面讲解 exec > file 的作用。运行 exec > file 之后，当前 shell 的任何命令的输出不再显示在终端屏幕，而是重定向到文件 file，试试就知道了：

```
$ exec > exec_out.txt
$ date                                # 命令 date 的结果没有显示在屏幕上
$ pwd                                 # 命令 pwd 的结果没有显示在屏幕上
$ echo "I study bash now."            # 命令 echo 的结果没有显示在屏幕上
```

以上运行 exec > exec_out.txt 之后，又执行了三条命令，屏幕上没有显示，应该是重定向到了文件 exec_out.txt，但不能查看该文件：

```
$ cat exec_out.txt
cat: exec_out.txt: input file is output file
```

不能查看的原因是：文件 exec_out.txt 是 cat 命令的输入参数，cat 命令试图将文件的内容显示出来，而当前 shell 的任何命令的输出都已重定向到文件 exec_out.txt，于是命令 cat exec_out.txt 试图将文件 exec_out.txt 的内容重定向到文件 exec_out.txt 中，导致 cat 命令的输入文件与输出文件是同一个文件（提示信息为 cat: exec_out.txt: input file is output file），所以，运行 cat exec_out.txt 失败。

那么，如何做才能查看文件 exec_out.txt 呢？有两种方法。第一种方法，重新打开一个 shell 窗口，即可查看，因为命令 exec > exec_out.txt 只对当前 shell 起作用；第二种方法，将当前 shell 的命令输出，重定向到终端屏幕，即/dev/tty，还原默认设置：

```
$ exec > /dev/tty
```

这时，就可以查文件 exec_out.txt 了。前面执行的三条命令的结果，都存入到文件 exec_out.txt 中：

```
$ cat exec_out.txt
Sun Mar   3 22:42:04 EST 2013
/home/user
I study bash now.
```

下面讲解 exec m< file 和 command <& m 的使用。命令 exec 3< exec_out.txt 是将文件 exec_out.txt 读入到文件描述符 3 中（注意 3 与<之间不能有空格）：

```
$   exec 3< exec_out.txt                # 文件 exec_out.txt 读入到文件描述符 3
```

下面的命令 cat <& 3 的意思是，将文件描述符 3 作为 cat 命令的输入，相当于运行 cat exec_out.txt（注意<与&之间不能有空格, &与 3 之间有无空格均可）：

```
$ cat <& 3                              # 相当于运行 cat exec_out.txt
Sun Mar   3 22:42:04 EST 2013
/home/user
I study bash now.
```

这时，进入/dev/fd 目录，可以查看到文件描述符 3：

```
$ cd /dev/fd
$ ls -l
total 0
lr-x------   1    user user 64 2013-03-04 10:12 0 -> /dev/pts/2
lrwx------   1    user user 64 2013-03-04 10:12 1 -> /dev/pts/2
lrwx------   1    user user 64 2013-03-04 10:12 2 -> /dev/pts/2
lrwx------   1    user user 64 2013-03-04 10:13 255 -> /dev/pts/2
lrwx------   1    user user 64 2013-03-04 10:12 3 -> /home/user/exec_out.txt
```

下面讲解 exec m> file 和 command >& m 的使用。命令 exec 5> exec_5.txt 的作用是，将写入文件描述符 5 的内容重定向到文件 exec_5.txt 中：

```
$ exec 5> exec_5.txt
```

下面的两条命令都重定向到文件描述符 5，所以两条命令的执行结果都不显示在屏幕上，而是重定向到文件描述符 5 对应的文件 exec_5.txt 之中：

```
$ pwd >& 5
$ echo "I study file descriptor" >& 5
```

严格地说，第一条命令 pwd 的结果重定向到了文件 exec_5.txt，第二条命令 echo 的结果追加重定向到了文件 exec_5.txt（如果接下来还有其他命令，命令的运行结果都会追加重定向到文件 exec_5.txt，而不是重定向到文件 exec_5.txt）。现在，查看文件 exec_5.txt，可见它存放着前面两条命令的执行结果：

```
$ cat exec_5.txt
```

```
/home/user                              # 命令 pwd 的执行结果重定向到 exec_5.txt
I study file descriptor                 # 命令 echo 的执行结果追加重定向到 exec_5.txt
```

下面讲解 exec n<& m 和 exec m<& -的使用。刚刚建立了文件描述符 5，命令 exec 6<& 5 创建了文件描述符 5 的副本——文件描述符 6：

```
$ exec 6<& 5
```

下面的 echo 命令重定向到文件描述符 6：

```
$ echo "I copy fd from 5 to 6" >& 6
```

此时，重定向到文件描述符 6 和重定向到文件描述符 5 效果一样，上面 echo 命令的执行结果存入了文件 exec_5.txt 中：

```
$ cat exec_5.txt
/home/user                              # 命令 pwd 的执行结果（重定向到文件描述符 5）
I study file descriptor                 # 命令 echo 的执行结果（重定向到文件描述符 5）
I copy fd from 5 to 6                   # 命令 echo 的执行结果（重定向到文件描述符 6）
```

exec m<& -的作用是关闭文件描述符 m。例如，将前面建立的文件描述符 5 关闭，命令 pwd 再重定向到文件描述符 5 时会失败：

```
$ exec 5<& -                            # 关闭文件描述符 5
$ pwd >& 5                              # 此时再使用描述符 5 将失败
bash: 5: Bad file descriptor
```

现在，文件描述符 5 没有了，但它的副本——文件描述符 6，仍然可以使用：

```
$ echo "file descriptor 6 exists" >& 6
$ cat exec_5.txt
/home/user                              # 命令 pwd 的执行结果（重定向到文件描述符 5）
I study file descriptor                 # 命令 echo 的执行结果（重定向到文件描述符 5）
I copy fd from 5 to 6                   # 命令 echo 的执行结果（重定向到文件描述符 6）
file descriptor 6 exists                # 命令 echo 的执行结果（重定向到文件描述符 6）
```

命令 exec >& -的作用是关闭标准输出。命令原本都可以正常屏幕输出，在关闭标准输出后，则不能正常运行，例如：

```
$ date
Mon Mar  4 11:46:29 EST 2013
$ exec >& -                             # 关闭标准输出
$ date                                  # date 命令运行失败
date: write error: Bad file descriptor
```

因为标准输出（屏幕）被关闭了，date 的结果没有地方可以显示。还原标准输出，命令就可以正常运行了：

```
$ exec > /dev/tty                       # 还原标准输出
$ date
```

```
Mon Mar  4 11:56:28 EST 2013
```

命令 exec <& -的作用是关闭标准输入，运行它的话，还没来得及看清楚，当前 shell 就退出了：

```
$ exec <& -
$ exit                                    # 自动运行 exit，退出 shell
```

前面的例子使用了文件描述符 3、5 和 6。那么，文件描述符的最大值是多少呢？带选项-n 的内置命令 ulimit 可以查询和设置这个值。运行 ulimit -n:

```
$ ulimit -n
1024
```

表示当前 shell 的可使用的文件描述符有 1024 个，那么取值范围就是 0 到 1023。下面设置其他整数值试试：

```
$ ulimit -n 500
$ ulimit -n
500                                       # 目前文件描述符最多可有 500 个
```

该值不是可以任意设置的，最大能设多少与具体的系统有关。例如，在某台计算机上，设置为 2048 时是成功的，设置为 2049 或者更大的数值时则失败：

```
$ ulimit -n 2049
bash: ulimit: open files: cannot modify limit: Operation not permitted
```

3.11　有关命令的进一步讨论

3.11.1　一行多命令

前面所有的例子，一行只有一条命令。实际上一行可以输入两条或者两条以上的命令，用分号隔开即可，命令的执行顺序和输入顺序相同。例如，将下面的三条命令放在一行，它们会依次运行：

```
$ date; echo "hello Mike"; pwd
Thu Oct 25 16:35:07 CST 2012              # date 命令的执行结果
hello Mike                                # echo 命令的执行结果
/home/user                                # pwd 命令的执行结果
```

3.11.2　将命令放在后台执行

前面讲过的命令看似都可以瞬间执行完毕，实际工作当中并不总是这样的。有的命令或者脚本需要很长时间才能执行完成，有的脚本甚至相当长一段时间一直处在运行状态，并且不需要用户监控它的执行状态，只需要查看最终的结果和中间的日志文件即可。这样的脚本或者命令应该放在后台执行。

假设某个目录存放了很多较大的文件，那么复制（cp）该目录需要一些时间，假设需要数分钟才能复制完该目录，那么在这段时间之内 shell 终端一直被 cp 命令占据着，不能执行别的命令。这时可以使用符号&，将 cp 命令放在后台执行，后台命令的格式为：

```
命令 &
```

那么复制内容很多的目录就可以运行 cp -r dir1 dir2 &，这样，shell 终端不会被 cp 命令一直占据着，马上就可以执行别的命令了（系统提示符立刻出现了）：

```
$ cp -r dir1 dir2 &
[1] 3073
$               # 系统提示符马上出现，可以执行别的命令了
```

在将一条命令放到后台运行时，会显示两个数字，一个是工作号，也叫作业号，另一个是进程号，上例中的工作号是 1，进程号是 3073。一条命令在后台运行时，用户可以通过某些命令操控它，有的命令使用的是后台命令的工作号，有的命令使用的是进程号。学习进程时，会有更详细的讨论。

把命令放在后台执行时，命令与&之间可以有空格，也可以紧挨着。建议命令与&之间有一个空格，可读性会好一些。

3.11.3　命令行的续行

前面讲的命令行的长度都不是很长，一行就够。例如：

```
$ echo "I study bash"
I study bash
```

假设这是一条长命令行，把这条命令分三行输入，看看会怎样（注意，下面例子中的>是系统的续行提示符，不是用户的键盘输入）：

```
$ echo "I
> study
> bash"
I
study
bash
```

分三行输入了命令，输出结果也是三行。如果希望输出结果是一行的话，在输入长命令时可在行尾用续行号\，也就是反斜线。例如：

```
$ echo "I \                    # 用续行号将一条命令分为三行
> study \
> bash"
I study bash                   # 输出为一行
```

用了续行号之后，分多行输入的命令，效果与在一行之内全部输入是一样的。

在讲解内置变量 PS1 的时候，提到过内置变量 PS2，其实，在本小节中，已经看到了

PS2 的值，其值为>。PS2 是命令续行提示符，默认值为大于号。下面将命令续行系统提示符 PS2 设置为百分号，再用续行号\将 date 命令“劈成两半”：

```
$ PS2=%
$ da\
%te
Thu Oct 25 21:02:01 CST 2012
```

可见，用续行号将 date 命令分为两行输入，和在一行之内输入的效果是一样的，“被劈开的”date 命令可正常运行。将 date 命令“劈成两半”时注意：da 与续行号\之间、\的后面、te 与系统续行提示符%之间，都不能有空格。

3.11.4 命令的补齐与命令历史

运行 echo 命令时，假如输入 ech 后，忘了后边的字母 o，这时，按一下〈Tab〉键，echo 命令被自动补齐了。再假如，本想运行 pwd 命令，但是输入 pw 后，忘了后边的字母 d，这时，按一下〈Tab〉键，系统没有将它自动补齐，这是因为以 pw 开头的命令有多个，再按一下〈Tab〉键，即，一共按两下〈Tab〉键，系统将所有以 pw 开头的命令列了出来供用户选择，例如：

```
$ pw                              # 输入 pw，再按两下〈Tab〉键
pwck       pwconv     pwd        pwdx       pwunconv
```

上面的例子说明命令名本身可以自动补齐，其实命令的参数也是可以补齐的。假设要删除某个以 lst 开头的文件，如果当前目录下只有一个以 lst 开头的文件，那么输入 rm lst，再按一下〈Tab〉键，文件名（rm 命令的参数）就会自动补齐；如果当前目录下有多个文件名以 lst 开头的文件，那么输入 rm lst，再按两下〈Tab〉键，系统会将所有以文件名 lst 开头的文件列出来供用户选择：

```
$ rm lst
lst000.txt    lst22.txt    lst_bak.txt  lst_name/    lst.txt
```

lst_name 的后面有斜杠，说明它是一个目录，后面没有斜杠的为文件。

在第 2 章讲过，运行如下命令可以将文件 a.txt 复制到/tmp/doc 目录下：

```
$ cp a.txt /tmp/doc
```

这条命令有两种可能的结果，当目录 doc 不存在时，a.txt 被复制为/tmp 目录下的文件 doc；当目录 doc 存在时，a.txt 被复制到/tmp/doc 目录下。

输入命令 cp a.txt /tmp/doc 后，按回车键之前，按〈Tab〉键，如果 doc 后面出现斜杠/（命令变为如下所示），说明目录/tmp/doc 存在。

```
$ cp a.txt /tmp/doc/
```

同样，对一个目录层数比较多、文件名字长的参数进行操作（复制或删除），例如，/home/maggie/test/script/project/log_scan/build_preparation.sh，全是键盘输入的话，容易出错且速度慢。如果一边输入，一边多按按〈Tab〉键，多使用系统的自动补齐功能，则可

以大大提高输入效率。

在工作中，刚刚执行过的命令有时需要再执行一次，特别是调试某个脚本和学习某条命令时，常常需要反复执行相同或者相近的命令。重新输入一次命令当然是可以的，但由于 shell 会把已经执行过的命令记录下来，所以只要按上、下方向键，就可以选择以前执行过的命令，而不必重新输入。

运行 Bash 的内置命令 history，可以看到历史命令。用户想参考老命令，或者想执行某条老命令的话，可以先执行命令 history。当然，并非所有的历史命令都一定被记录下来，这受到某些 Bash 的内置变量的控制，在 4.14 节会讲到。

3.11.5　操作名字含空格的文件

Windows 下的文件名可以有空格，Linux 下的文件名也可以包含空格。产生一个新文件的时候最好避免让它的名字包含空格，因为操作一个名字带空格的文件是不方便的。例如有个文件叫做“home work”，不能使用 cat home work 查看它，因为 cat home work 的意思是查看文件 home 和文件 work：

```
$ cat home work
cat: home: No such file or directory          # 文件 home 不存在
cat: work: No such file or directory          # 文件 work 不存在
```

操作名字带空格的文件，要用引号（单引号和双引号均可）括起来，或者在文件名当中的空格前面加反斜杠\，如有多个连续的空格，每个空格前面都要加一个反斜杠：

```
$ cat "home work"
this file name includes space
file name is 'home work'
$ cat home\ work
this file name includes space
file name is 'home work'
```

用 rm "home work"或者 rm home\ work 删除它；用 touch "home work"或者 touch home\ work 可以产生一个名字带空格的新文件 home work。可见，对名字带空格的文件进行操作确实是不方便的，所以最好避免产生名字包含空格的文件。

3.11.6　操作名字首字母为减号的文件

如果遇到一个名字的首字母为减号的文件，如何删除它呢？试试用命令 rm 删除它：

```
$ rm -a.txt
rm: invalid option -- 'a'
Try `rm ./-a.txt' to remove the file `-a.txt'.
Try `rm --help' for more information.
```

因为 Linux 命令的选项是以减号开头的，所以执行 rm -a.txt 时，减号后面的字符会被认为是选项，而 rm 命令没有这样的选项，故运行出错。仔细看上面提示信息后两行，再运行 rm --help 看帮助，就知道办法了。在名字首字母为减号的文件名前加./或者“-- ”（两个减号和一个空

格），就可以正常操作该文件。即运行 rm ./-a.txt 或者 rm -- -a.txt 可以删除-a.txt。

产生这样的文件的方法和删除它的方法是一致的。如命令 touch -- -b.txt 和 touch ./-b.txt 可以产生空文件-b.txt；命令 gedit，nedit 和 vi 后面跟-- -c.txt 或者./-c.txt 可以编辑文件-c.txt；命令 cp z.txt -- -x.txt 和 cp z.txt ./-x.txt 的作用是将文件 z.txt 复制为-x.txt。

把-a.txt 变为./-a.txt 可以解决问题，是因为这样就避免了减号出现在一个参数的最前面，减号后的字符就不会被误认为是选项；把-a.txt 的前面加上--可以解决问题，是因为 Bash 规定，命令行中--后面的东西不被认为是选项，而被认为是文件名或者其他参数，见表 10-1。

将一个文件复制为某个目录下的文件名首字母为减号的文件时，或者删除某个目录下的文件名首字母为减号的文件时，不需要使用./或者--，例如：

```
$ cp z.txt /tmp/-ss.txt
$ rm /tmp/-ss.txt
```

因为这时命令（如 rm /tmp/-ss.txt）中的减号不出现在参数的最前面。

本节讲的问题，在工作中比较少见，倒是在面试的时候很可能遇到。但是，通过学习本节内容，可以加深对某些概念的理解。

3.11.7 Windows 与 Linux 文件格式的互转

有时在 Windows 下编写好一个脚本，复制到 Linux 里不能正常运行，表面上也看不出脚本的语法问题，但就是不能正常运行；有时 Linux 下的某个脚本本来可以正常运行，可是从 Windows 的某处（从文件或邮件里）复制了几句命令，粘贴到 Linux 的这个脚本里，保存后，这个脚本也变得不能正常运行。出现这两种情况（实际上是一种情况：Linux 中的脚本包含了来自 Windows 的内容）的基本原因是两种系统的文件格式不同。

例如，在 Windows 下用记事本编辑好如下脚本，并传到了 Linux 里（为保证传入 Linux 的过程中，文件的格式不变，可以先在 Windows 下将脚本压缩为.zip 文件，传入 Linux 后再用 unzip 命令解压缩）：

```
$ cat from_win.sh
#!/bin/bash
echo "I am from Windows."
```

用 chmod +x 命令增加其执行权限之后，发现脚本不能运行：

```
$ chmod +x from_win.sh
$ from_win.sh
bash: ./from_win.sh: /bin/bash^M: bad interpreter: No such file or directory
```

在 Windows 下，文件的换行符为\r\n，而在 Linux 下，它是\n。这个差异造成了脚本错误。当一个 Linux 下的文件包含了来自 Windows 的内容，或者这个文件就来自 Windows，用命令 dos2unix <文件名>转换其格式，文件格式差异的问题就解决了。例如，转换格式后，脚本 from_win.sh 就可以正常执行了：

```
$ cp from_win.sh from_win_2_Linux.sh          # 为便于对比，先复制为 from_win_2_Linux.sh
$ dos2unix from_win_2_Linux.sh                # 用 dos2unix 命令转换格式
```

```
$ from_win_2_Linux.sh                    # 可以正常运行了
I am from Windows.
```

如果读者感兴趣，可以用命令 ls -l 查看格式转换前后脚本的大小，可以发现字节数不同。

同样，如果 Linux 下的某个文件要复制到 Windows 下使用，最好先执行命令 unix2dos〈文件名〉，然后再将该文件复制到 Windows，会避免潜在的麻烦。

为进一步理解上面的内容（或者，如果没有成功看到上面的现象的话），也可以在 Linux 下找一个原本能够正常运行的脚本，用命令 unix2dos〈脚本名〉改变其格式，再执行脚本试试，然后用命令 ls -l 查看格式转换前后脚本的字节数。

3.11.8 小括号与大括号中的命令

一条或者多条命令，可以放在小括号之中，也可以放在大括号之中。放在小括号之中时，命令是在一个子 shell 里面运行；放在大括号之中时，命令是在当前 shell 里面运行。看一个例子：

```
$ x=5
$ (x=10)
$ echo $x
5
$ { x=10;}
$ echo $x
10
```

因为(x=10)是在子 shell 里面给 x 赋值为 10，并不影响当前 shell 中的 x 的值，所以第一个 echo 命令显示 5。而{ x=10;}是当前 shell 里面给 x 赋值为 10，所以第二个 echo 命令显示 10。使用大括号的时候注意，左大括号的右面必须有空格，右大括号的左面必须有分号。

再看一个例子，在小括号里面改变路径不影响当前 shell 的路径：

```
$ pwd
/home/user                   # 确定当前 shell 的当前路径是/home/user
$ (cd /tmp; pwd)             # 在小括号里面改变路径，再显示路径
/tmp                         # 子 shell 的路径变为了/tmp
$ pwd
/home/user                   # 当前 shell 的路径不受影响，仍然是/home/user
```

在大括号里面改变路径，就是改变当前 shell 的路径：

```
$ pwd
/home/user
$ { cd /tmp; pwd; }
/tmp
$ pwd
/tmp                    # 当前 shell 的路径变为/tmp
```

3.11.9 子 shell

小括号里面的命令是在当前 shell 的子 shell 里面执行，当前 shell 是它的子 shell 的父

shell，它们之间的关系如图 3-4 所示：

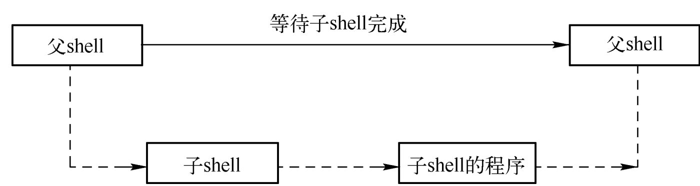


图 3-4　子 shell 与父 shell

子 shell 与父 shell 有各自的进程 ID，运行 ps 可以查看进程。例如，运行 pwd 和 ps 命令：

```
$ pwd; ps
/home/user                                      # 命令 pwd 的执行结果
  PID    TTY        TIME      CMD               # 命令 ps 的执行结果
 3111    pts/3      00:00:00  bash              # 当前 shell 的进程 ID 为 3111
 3195    pts/3      00:00:00  ps                # 命令 ps 的进程 ID 为 3195
```

如果把这两条命令放在小括号里面也就是放在一个子 shell 里面运行，ps 命令将显示子 shell 和其父 shell 的进程 ID：

```
$ (pwd; ps)
/home/user                                      # 命令 pwd 的执行结果
  PID    TTY      TIME      CMD                 # 命令 ps 的执行结果
 3111    pts/3    00:00:00  bash                # 父 shell 的进程 ID 仍为 3111
 3196    pts/3    00:00:00  bash                # 子 shell 的进程 ID 为 3196
 3197    pts/3    00:00:00  ps                  # 命令 ps 的进程 ID 变为 3197
```

当前 shell（小括号里面的子 shell 的父 shell）的进程 ID 仍为 3111，只要当前 shell 不被关闭，即，不要关闭当前 shell 又重新打开一个 shell 窗口，那么它的进程 ID 不变。

再运行一次：

```
$ (pwd; ps)
/home/user                                      # 命令 pwd 的执行结果
  PID    TTY      TIME      CMD                 # 命令 ps 的执行结果
 3111    pts/3    00:00:00  bash                # 父 shell 的进程 ID 仍为 3111
 3212    pts/3    00:00:00  bash                # 子 shell 的进程 ID 变为 3212
 3213    pts/3    00:00:00  ps                  # 命令 ps 的进程 ID 变为 3213
```

可见，父 shell 的进程 ID 不变，子 shell 和子 shell 中的 ps 命令的进程 ID 变了，同样的命令，每运行一次，进程 ID 都不一样。关于 ps 命令，在第 9 章有较详细的讲解。

第 4 章　变量与数组

Bash 没有 C/C++、Pascal、Fortran 等高级语言那样丰富的数据类型，可以认为它只有整数型和字符串型两种基本数据类型。但是，严格地说，Bash 变量本质上都是字符串。数组就是相同数据类型的元素的集合，Bash 支持数组。本章介绍了普通变量、数组、环境变量和内置变量，并讲解了变量赋值和运算的相关命令。

4.1　变量的定义与清除

在 C/C++，Pascal 等语言中，首先要声明变量的类型，才可以使用，而 Bash 无需事先声明其变量的类型就可以直接赋值。引用变量时，前面需加美元符$，如：

```
$ age=28
$ echo $age
28
$ sentence="We are friends"
$ echo $sentence
We are friends
```

变量前面不加$，看看会怎么样：

```
$ echo age
age                    # 不加$，则显示字符串 age，而不是变量 age 的值
$ echo sentence
sentence
```

由于 Bash 只有字符串类型，无论将一个整数赋给一个变量，还是将一个字符串赋给一个变量，Bash 都把它当字符串处理。区别是，当把整数赋给一个变量时，该变量具有了整数的属性，可以进行整数运算，包括四则运算、大小关系比较运算等。

定义变量时，注意变量名字必须是字母数字下画线组合，首字符必须是字母或者下画线。下面的例子，定义变量用短线（减号）时，遇到错误，将短线改为下画线即可：

```
$ old-man-age=89
old-man-age=89: command not found
$ old_man_age=89
$ echo $old_man_age
89
```

还要注意，变量定义时，等号两边不能有空格或者〈Tab〉键，否则变量定义不会成功。

```
$ age= 28
bash: 28: command not found            # 等号右边有空格时遇到错误
$ age =28
bash: age: command not found           # 等号左边有空格时遇到错误
```

没有被定义的变量，也可以直接引用，不过这时没有值。下面例子中的 project_name，若是没有定义就显示其值，将显示空行。

```
$ echo $project_name
                              # 空行
```

也可以将变量赋为空，等号的右边不输入任何东西，即在等号右边直接输入回车，就是将变量赋为空。例如：

```
$project_name=[回车]
$ echo $project_name
                              # 空行
```

引用一个被赋为空的变量，和直接引用一个未被定义的变量，效果是相同的，都得到空。但是这二者还是有区别的：被赋为空的一个变量，在变量表里是找得到的，只不过它的值是空的；未被定义的变量，在变量表里是找不到的。运行内置命令 set，可以查看当前 shell 的所有的变量和函数定义。

用内置命令 unset 清除变量的定义，例如：

```
$ unset age
$ unset sentence
```

清除后，再试着显示其值，将得到空行：

```
$ echo $age
                              # 空行
$ echo $sentence
                              # 空行
```

4.2　字符串定义及单双引号与大括号的使用

定义字符串的时候，如果字符串包含空格，则需要使用引号。

```
$ b=Good afternoon                # 试图将字符串 Good afternoon 赋给 b
bash: afternoon: command not found
```

上面的 b 没有赋值成功，是因为等号右边的字符串有空格。在字符串两边加上引号或者在空格前面加反斜杠，就可以成功赋值。

```
$ a="Good morning"
$ b='Good afternoon'
```

```
$ c=Good\ night
$ echo $a, $b, $c
Good morning, Good afternoon, Good night
```

从上面的例子看，似乎单引号与双引号的作用是等同的，实际不然。

```
$ a=Good
$ b="$a morning"
$ echo $b
Good morning
$ b='$a morning'
$ echo $b
$a morning
```

上面的例子中，先将 a 定义为 Good，再将 b 定义为"$a morning"时（两边是双引号），a 的值会被代入，或者说 a 的值被展开，b 的值就成为 Good morning。如果将 b 定义为'$a morning'时（两边是单引号），a 的值不会被代入，单引号之间的内容被原封不动地赋给了 b，b 的值成为了$a morning。

下面将 B 定义为空格加 morning，将 A 定义为 Good：

```
$ B=" morning"
$ A=Good
```

可以这样定义 C，使其值为 Good morning：

```
$ C=$A$B
$ echo $C
Good morning
```

或者，这样定义：

```
$ C=$A" morning"
$ echo $C
Good morning
```

如果希望 C 的值为 Good_morning（注意，Good 与 morning 之间是下画线），是不是可以这样赋值呢？

```
$ A=Good
$ C=$A_morning
$ echo $C
                        # 空行
```

可见，C 的值是空。这是因为将 C 定义为$A_morning 时，A_morning 的值会被代入，虽然 A 是个已经有值的变量，但 A_morning 是一个未定义的变量，执行 C=$A_morning 之后，C 被赋值为空，这不是用户想看到的结果。将变量 A 的两边加上大括号就可以解决这个问题：

```
$ C=${A}_morning
$ echo $C
Good_morning
```

A 两边有了大括号，A 就会被认为是一个变量名，系统不会再认为 A_morning 是一个变量名，A 的值被代入，C 的值就成为了 Good_morning。大括号是变量名字的定界符。

```
$ project_name=[回车]
$ echo "xx${project_name}yy"
xxyy                      # xx 和 yy 之间没有空格
```

上面的例子，project_name 被定义为空，所以 echo 命令中的 xx 与 yy 之间实际上没有任何东西，echo 命令的执行结果就是 xxyy。

如果希望将 project_name 定义为一个空格，需要将空格用引号括起来：

```
$ project_name=' '
```

下面的 echo 命令的执行结果中有一个空格介于 xx 和 yy 之间：

```
$ echo "xx${project_name}yy"
xx yy                     # 有一个空格介于 xx 和 yy 之间
```

如果空格不被引号括起来，即在等号右边输入一个空格再回车的话，project_name 将被定义为空：

```
$ project_name=[空格][回车]
$ echo "xx${project_name}yy"
xxyy                      # xx 和 yy 之间没有空格
```

可见，将空格赋值给变量时，需要将空格用引号括起来，否则变量的值为空而不是空格。

一般情况下，系统认为单引号或者双引号应该成对出现，例如，输入 echo 'Good 回车，系统不认为命令行输入完成，因此给出命令续行提示符>，直到遇到另一个单引号，系统才认为命令行输入完成。例如：

```
$ echo 'Good              # 键盘输入'Good
> morning                 # 键盘输入 morning
> Mike'                   # 键盘输入 Mike'
Good
morning
Mike
```

如果想显示 It's a book 这句话怎么办？先试着运行一下：

```
$ echo It's a book
>
```

遇到了命令续行提示符>，系统在等待另一个单引号的出现。把含单引号的句子用双引号括起来，就可以正常显示了：

```
$ echo "It's a book"
It's a book
```

同样，把含双引号的句子用单引号括起来，即可以正常显示：

```
$ echo 'double quotation mark is:", right'
double quotation mark is:", right
```

或者，在引号前面加反斜杠，单独的引号可以显示出来，而无需成对出现：

```
$ echo It\'s a book
It's a book
$ echo double quotation mark is:\", right
double quotation mark is:", right
```

大括号除了可作为变量名的定界符，还有扩展功能。使用该功能时，需将一些字符串以逗号间隔，放在一对大括号中间。左大括号的前面可以有别的字符串，叫做前导；右大括号的后面也可以有别的字符串，叫做后继。大括号中的字符串与它们可以组成新的字符串，并保留原来的顺序。例如：

```
$ echo a{d,c,b}e
ade ace abe
```

大括号扩展可以嵌套。例如：

```
$ echo a{d,c,b{x,yy,zzz}}e
ade ace abxe abyye abzzze
```

使用大括号扩展，有时可以使命令更快捷。例如，同时创建三个子目录：

```
$ mkdir -p /tmp/mail/{jack,mike,rose}
```

这条命令比 mkdir -p /tmp/mail/jack /tmp/mail/mike /tmp/mail/rose 紧凑，而且更容易让人看出来这三个子目录都在/tmp/mail 下面。

4.3 将命令执行结果赋给变量——反引号与$()

可以将一条命令或者某个脚本的执行结果赋给变量，有两种方法。

一种方法是，用反引号将命令括起来，格式为：

```
变量=`命令`
```

另一种方法是，用美元符和小括号将命令括起来，格式为：

```
变量=$(命令)
```

例如，将命令 date 的运行结果赋给变量 A 和 B：

```
$ A=`date`
$ echo $A
Sat Apr 16 22:51:26 CST 2011                # 变量 A 存放了 date 命令的执行结果
$ B=$(date)
$ echo $B
Sat Apr 16 22:57:47 CST 2011                # 变量 B 存放了 date 命令的执行结果
```

也可以将命令的引用直接放在另一条命令里面：

```
$ echo Now, time is $(date)
Now, time is Sat Apr 16 22:59:28 CST 2011
$ echo Current directory is $(pwd)
Current directory is /home/user
$ rm -f `find -name "*.txt"`                # 找到.txt 文件后直接删除
```

输出内容为多行的命令，其输出也可以赋给变量。例如，ls -l 命令的输出一般是多行的，将它的输出赋给 a：

```
$ a=$(ls -l)
$ echo $a
total 24 -rw-r--r-- 1 user user 121 2012-08-17 03:04 1.pl drwxr-xr-x 2 user user 4096 2012-08-17
03:04 bin -rw-r--r-- 1 user user 64 2012-08-17 03:04 cronjob.txt -rw-r--r-- 1 user user 4253
2012-08-17 03:04 makefile -rw-r--r-- 1 user user 3044 2012-08-17 03:04 mkview.pl -rw-r--r-- 1 user
user 171 2012-08-17 03:04 mt.sh
```

运行 echo $a 时，ls -l 命令结果并没有像预期的那样多行显示，而是在一行上显示了。这是因为命令 echo $a 将换行符丢掉了。将变量用双引号括起来，可以保留换行符：

```
$ echo "$a"
total 24
-rw-r--r--   1 user user   121    2012-08-17 03:04 1.pl
drwxr-xr-x  2 user user 4096    2012-08-17 03:04 bin
-rw-r--r--   1 user user    64    2012-08-17 03:04 cronjob.txt
-rw-r--r--   1 user user 4253    2012-08-17 03:04 makefile
-rw-r--r--   1 user user 3044    2012-08-17 03:04 mkview.pl
-rw-r--r--   1 user user   171    2012-08-17 03:04 mt.sh
```

现在可以顺便再解释一下第 3 章提到的“黑洞”的作用。例如，运行 a=`script_1.sh`可以将脚本 script_1.sh 的执行结果赋给变量 a。在脚本 script_1.sh 当中，可能调用了很多命令甚至调用了很多别的脚本，这些命令和脚本很可能大多有正常的输出和异常（错误信息）的输出。假设 script_1.sh 一定有执行结果，而且只需要用变量 a 记录这个结果，并不关心中间的输出（包括正常的输出和异常的输出），那么，编写脚本 script_1.sh 的时候，脚本当中某些命令的输出就可以重定向到“黑洞”/dev/null。

反引号和$()的作用相同，但从长远的发展看，$()会用得越来越多。老脚本当中反引号用得多一些，新脚本当中$()用得多一些。$()的优点是容易嵌套，看下面的例子：

```
$ time=$(echo $(date))
$ echo $time
Fri Dec 28 19:15:44 CST 2012
```

如果将上面命令中的$()换为反引号，再试试：

```
$ time=`echo `date``
$ echo $time
date
```

结果不是期望的那样，因为，系统会将`echo `date``的前面两个反引号配对，后面两个配对。这样，前面两个反引号之间为一个空 echo 命令，后面两个反引号之间为一个纯粹的空命令，所以，命令 time=`echo `date``就相当于 time=date，把字符串 date 赋给了 time。

可以将反引号或者$()放在单双引号之间。放在单引号之间时，原样显示出来，反引号之间的命令或者$()中的命令不执行：

```
$ echo '`date`'
`date`
$ echo '$(date)'
$(date)
```

放在双引号之间时，反引号之间的命令或者$()中的命令被执行：

```
$ echo "`date`"
Thu May 30 21:05:53 EDT 2013
$ echo "$(date)"
Thu May 30 21:06:43 EDT 2013
```

放在双引号之间时，如果希望原样显示出来，需要在反引号和美元符左边加反斜杠。加反斜杠就可以去除反引号和美元符的作用，让其原样显示出来：

```
$ echo "\`date\`"
`date`
$ echo "\$(date)"
$(date)
```

4.4 键盘输入变量值——内置命令 read

使用内置的 read 命令，通过键盘输入可以对变量赋值:

```
$ read var
```

```
45                         # 键盘输入 45
$ echo $var
45                         # 命令 echo 显示出的 45
$ read var
"how are you?"             # 键盘输入"how are you?"
$ echo $var
"how are you?"             # 命令 echo 显示出的"how are you?"
```

上面的例子说明，对字符串赋值时，如果带上引号，则引号本身被认为是字符串的一部分。

如果输入字符串值的时候有连续多个空格，那么显示字符串的值时，用不用双引号效果是不一样的。例如：

```
$ read var
how        are        you?        # 键盘输入时，单词间有多个空格
$ echo $var
how are you?                      # 没有双引号，显示时，单词之间只剩一个空格
$ echo "$var"
how        are        you?        # 用双引号，显示时，空格数不变
```

read 命令可以一次读取多个变量的值：

```
$ read var1 var2 var3
78 56 100
$ echo $var1, $var2, $var3
78, 56, 100
```

read 命令后面不跟变量名时，输入的值被存入内置变量 REPLY 中：

```
$ read                     # 命令 read 后没有跟变量
Morning                    # 键盘输入 Morning
$ echo $REPLY
Morning                    # 内置变量 REPLY 的值为 Morning
$ read
78                         # 键盘输入 78
$ echo $REPLY
78                         # 内置变量 REPLY 的值为 78
```

read 命令的-p 选项，用于在键盘输入前给用户提示（prompt），格式为：

```
read -p 提示信息 变量
```

例如：

```
$ read -p "please input the phone number: " phone_number
please input the phone number: 88888888        # 键盘输入电话号码 88888888
$ echo $phone_number
88888888
```

read -p 命令中的提示信息也可以是一个字符串变量。在写脚本、要求用户键盘输入时，尽量多用选项-p，可以给用户明确的输入提示。或者在 read 命令之前用 echo 命令给用户显示提示也是很好的做法。例如：

```
echo "please input your phone number:"
read phone_number
```

命令 read 的选项-t 后面跟数字，单位为秒，可以进行计时输入。例如，下面的命令给了用户 5 秒钟时间，5 秒内没有键盘输入的话，read 命令超时（timeout）退出，退出状态为非 0（失败）：

```
$ read -t 5 -p "please input your name:" name
please input your name: $ echo $?
142
```

选项-t 后面的数字也可以是小数。如果已经设置了内置变量 TMOUT 的值（见表 4-3），那么这个值将作为 read 命令的默认超时时间。

选项-n 用来计数输入的字符，当输入的字符数目达到预定数目时，自动退出，并将输入的数据赋值给变量。例如，下面命令中的-n 1 表示：只接受一个输入字符。准备输入 Yes 给变量 answer，当输入完字母 Y 时，自动退出（不等到按其他键和回车键），变量 answer 被赋值为 Y：

```
$ read -n 1 -p "Do you accept it [Y/N]? " answer
Do you accept it [Y/N]? Y $ echo $answer
Y
```

选项-s 使得 read 命令进入“静音模式（silent）”，即用户的键盘输入不显示在屏幕上。最典型的例子就是在输入密码的时候。运行下面的命令，输入的密码是看不见的，输入完成后，再显示变量 my_passwd 的值才能看见密码：

```
$ read -s -p "please input password: " my_passwd
please input password: $ echo $my_passwd
w%D369a
```

read 命令还可以使用文件描述符读入变量的值，选项及参数为-u fd，fd 代表文件描述符，使用文件描述符给变量赋值的方法不是特别常用。

4.5　整型变量运算

整型的计算要放在双括号之间或者放在内置命令 let 的后面，例如：

```
$ x=20 y=30              # 同一行对多个变量赋值，用空格分隔
$ ((w=x+y))
$ echo $w
50
```

```
$ let k=x*y
$ echo $k
600
```

如果整型的计算没有放在双括号之间，也没有放在 let 的后面，会怎样呢？

```
$ u=x-y
$ echo $u
x-y
```

u 被定义为了字符串 x-y。将其放在 let 的后面，或者放入双括号里，就得到想要的结果了：

```
$ ((u=x-y))
$ echo $u
-10
```

也可以直接进行整数运算，赋值给某个变量：

```
$ let m=600/10
$ echo $m
60
$ n=$((80/10))
$ echo $n
8
```

还可以直接对加减乘除式子进行计算，并显示其值。遵从由左至右、先乘除后加减、括号里面最优先的四则运算法则，如：

```
$ echo $(((200+70)/(4+6)))
27
$ echo $((5+3*7))
26
```

两个不能整除的整数相除时，原本应该得到小数，但小数点后面的部分不被保留。这是由于 Bash 只有整型数，没有实型数，小数点后面的部分会被丢弃而不是四舍五入。下例中，原本 50 除以 20 为 2.5，50 除以 30 为 1.667，但实际上分别得到整数值 2 和 1，小数点后面的部分会被丢弃：

```
$ echo $((50/20))
2
$ echo $((50/30))
1
```

%为求余运算符，下例中 50 除以 30、除以 20、除以 16 的余数分别为 20、10 和 2：

```
$ echo $((50%30))
20
```

```
$ echo $((50%20))
10
$ echo $((50%16))
2
```

**为幂运算符，下例中打印出来 9 的平方值和 9 的立方值，分别是 81 和 729：

```
$ ((y=9**2))
$ ((z=9**3))
$ echo $y, $z
81, 729
```

与 C 语言一样，Bash 支持++运算和--运算。++和--分别为变量的自加和自减运算。当++位于变量的右边时为“先用后加”。看了下例就明白了，执行 echo $((i++))时，“先用”，就是先使用 i 的值，显示 9，“后加”，i 的值自加 1，再执行 echo $i，显示 10：

```
$ i=9
$ echo $((i++))
9
$ echo $i
10
```

当++位于变量的左边时为“先加后用”。看下例，执行 echo $((++i))时，“先加”，就是先将 i 的值加 1，“后用”，而后再显示 i 的值，则显示 10，接着再执行 echo $i，i 的值已经是 10 了，当然还显示 10：

```
$ i=9
$ echo $((++i))
10
$ echo $i
10
```

同样，当--位于变量的右边时为先用后减；当--位于变量的左边时为先减后用。++和--是单目运算符，+，-，*，/，%等是双目运算符。

与 C 语言一样，Bash 支持+=、-=、*=、/=和%=等运算。见下例，k+=1 相当于 k=k+1，m-=k 相当于 m=m-k：

```
$ k=10 m=20
$ ((k+=1))
$ echo $k
11
$ let m-=k
$ echo $m
9
```

Bash 还支持位运算等多种运算。Bash 支持的运算符见表 4-1。

表 4-1　Bash 运算符

运　算　符	意　　义
+ - * / % **	加、减、乘、除、取余数、幂
! ~	逻辑取反、位取反
++ --	变量自加、自减
< <= > >=	大小比较
== !=	相等与不等
& | ^	位的与、或、异或
<< >>	向左位移、向右位移
&& ||	逻辑与、逻辑或
? :	条件运算符，也叫三目或者三元运算符
+= -= *= /= %= &= ^= <<= >>= |=	赋值运算符

像在 C 语言里面一样，关系运算（<、<=、>、>=、==、!=）的结果，如果是 0，表示假，1 表示真。例如，4>5 是假的，8>=7 是真的：

```
$ echo $((4 > 5))
0
$ echo $((8 >= 7))
1
```

关系运算的结果，0 表示假，1 表示真；Linux 命令执行成功则退出状态为 0，非 0 意味执行失败。这句话中的两个 0 代表的意思是不一样的，要注意区分。

一个假与一个真的逻辑或的结果为真：

```
$ echo $(( 4 > 5 || 8 >= 7))
1
```

下面讲解表 4-1 中的三目运算符，它的语法格式为：

条件? 结果 1: 结果 2。

问号前面是判断条件，如果条件满足时取结果 1，不满足时取结果 2。看下面的例子，2>1 是成立的，三目运算结果为 5，所以 a 的值为 5：

```
$ a=$((2>1?5:8))
$ echo $a
5
```

下面的例子，2>3 不成立，三目运算结果为 8，所以 b 的值为 8：

```
$ b=$((2>3?5:8))
$ echo $b
8
```

以上在双小括号里面的计算，也可以放在中括号里面进行。例如：

```
$ echo $[80*6]
480
$ echo $[(200+70)/(4+6)]
27
$ echo $[2>3 ? 5 : 98]
98
```

Bash 运算符的用法与 C 语言的基本上是一致的，这里就不一一列举了。实际上，数值运算不是 Bash 的强项，一般的 Bash 脚本中只有数值不大的整数运算，如控制循环次数，对某个操作的计数等。如果要进行整数运算或者精度要求不高的浮点数运算，可以使用 C 或者 Fortran 语言；如果要进行精度要求较高的浮点数运算，应该使用 Fortran、MATLAB 或者 Mathematica。

4.6 浮点运算

如要在 shell 里面进行浮点运算，可以使用工具 bc 或者 awk。bc 是 Linux 下的一种高精度计算语言，这里不详细展开讲，只举几个简单的例子。

在 bc 中，scale 的值表示小数点后的保留位数，下例将 scale=3 和 13/2 通过管道传给 bc，13/2 等于 6.5，小数点后保留 3 位，就是 6.500：

```
$ echo "scale=3; 13/2" | bc
6.500
```

小数点后保留 8 位，就是 6.50000000：

```
$ echo "scale=8; 13/2" | bc
6.50000000
```

在 bc 中，sqrt(a)表示 a 的平方根。小数点后保留 8 位，2 的平方根：

```
$ echo "scale=8; sqrt(2)" | bc
1.41421356
```

也可以直接运行 bc。进入 bc 后，直接输入要计算的式子，即可得到结果。刚刚进入 bc 时，默认会显示版本信息。

```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
scale=3            # 小数点后保留 3 位
sqrt(3)
1.732              # 3 的平方根等于 1.732
4^3
64                 # 4 的 3 次方等于 64
```

```
3*4+5/2
14.500            # 先乘除后加减，3*4+5/2 等于 14.500
quit              # 运行 quit，退出 bc
```

4.7　定义只读变量命令 readonly 和 declare -r

用内置命令 readonly 可以给变量设置只读属性，定义只读变量。定义为只读之后，变量不允许再被赋值：

```
$ readonly x=9
$ x=10
bash: x: readonly variable          # 不能再给只读变量 x 赋值
$ x=9
bash: x: readonly variable          # 再次给它赋值为 9 也不可以
$ c=20
$ readonly c
$ c=30
bash: c: readonly variable          # 不能再给只读变量 c 赋值
$ readonly a
$ a="I study bash"
bash: a: readonly variable          # 不能给只读变量 a 赋值
$ echo $a
                                    # 空行，因为 a 没有值
```

只读变量也不允许被取消定义：

```
$ unset a
bash: unset: a cannot unset: readonly variable
```

用带选项-r 的内置命令 declare 或者 typeset 也可以定义只读变量。

```
$ declare -r Z="we are friends"
$ Z=20
bash: Z: readonly variable          # Z 是只读变量，不能再给它赋值
$ echo $Z
we are friends
```

运行 readonly -p 或者 declare -rp 可以显示（print/display）所有的只读变量。

内置命令 declare 和 typeset 功能完全一致，但是 typeset 似乎有被淘汰的趋势，所以，应尽可能多地使用 declare。

4.8　定义整型变量命令 declare -i

用带选项-i 的 declare 命令，可以使变量具有整数（integer）属性。Bash 本质上只有字

符型变量，但通常也可以说：定义整型变量。

例如，希望得到 45+20 的整数运算结果：

```
$ M=45+20
$ echo $M
45+20
```

执行 M=45+20 之后，M 的值为字符串 45+20，而不是希望得到的 65。执行 M=$((45+20)，或 let M=45+20 才能得到 65。如果事先声明变量是整型数，就不必使用$((…))或者 let。见下例：

```
$ declare -i K
$ K=45+20
$ echo $K
65
```

再看一个例子。C 被声明为整型变量，所以 C 的值为 15+30=45；D 没有被声明为整型，所以 D 的值是字符串 15+30，而不是期望得到的 45：

```
$ A=15 B=30
$ declare -i C=$A+$B
$ D=$A+$B
$ echo C=$C, D=$D
C=45, D=15+30
```

如果整型数被赋值为字符或不带空格的字符串时，系统不会报错，并且整型变量实际被赋值为 0：

```
$ declare -i M          # 声明整型变量 M
$ M="web"               # 变量 M 被赋值为无空格的字符串
$ echo $M
0                       # 显示 M 的值为 0
```

整型数被赋值为带空格的字符串时，系统将报错：

```
$M="web site"           # 变量 M 被赋值为带空格的字符串，出错
bash: web site: syntax error in expression (error token is "site")
```

前面提到的数都是十进制数，Bash 支持其他进制的数，支持的范围是从二进制到六十四进制。非十进制数赋值的格式为：

变量=基数#数值

基数的取值范围为 2 到 64，常用的有二进制、八进制、十六进制等。例如：

```
$ declare -i x
$ x=2#110          # 二进制的 110 赋给 x
$ echo $x
6                  # 二进制的 110 就是十进制的 6
```

```
$ x=8#16          # 八进制的 16 赋给 x
$ echo $x
14                # 八进制的 16 就是十进制的 14
$ x=16#2A         # 十六进制的 2A 赋给 x
$ echo $x
42                # 十六进制的 2A 就是十进制的 42
```

以 0 开头的数表示八进制数，以 0x 开头的数表示十六进制数，这一点与 C 语言一致。例如：

```
$ x=025           # 八进制的 25 赋给 x
$ echo $x
21                # 八进制的 25 就是十进制的 21
$ x=0x25          # 十六进制的 25 赋给 x
$ echo $x
37                # 十六制的 25 就是十进制的 37
```

赋值时，某一位数大于基数的话，赋值将失败。例如：

```
$ x=029           # 以 0 开头的为八进制数，数字只能取 0 到 7，9 超出基数，赋值失败
bash: 029: value too great for base (error token is "029")
$ x=16#2G         # 十六进制数字只能取 0 到 9、A 到 F，G 超出基数，赋值失败
bash: 16#2G: value too great for base (error token is "16#2G")
```

4.9　数组

Bash 只支持一维数组，不支持多维数组。Bash 数组的下标默认从 0 开始，而不是从 1 开始。用小括号把所有元素括起来，元素之间以空格隔开，就可以定义数组。下面定义了数组 y，一共 3 个元素：

```
$ y=(aa bb cc)
```

显示 3 个元素的值，即，第 0 个、第 1 个、第 2 个元素的值：

```
$ echo ${y[0]}, ${y[1]}, ${y[2]}
aa, bb, cc
```

使用带选项-a 的 read 命令，可以创建数组（array）并通过键盘输入数组元素的值，数组下标从 0 开始并连续。例如，下面的命令创建了数组 array_1，一共 4 个元素，数值分别是 12、34、56、78：

```
$ read -a array_1
12 34 56 78       # 键盘输入这 4 个数，以空格分隔
```

通过${数组名[*]}可以得到数组的所有元素，通过${#数组名[*]}可以得到数组元素的个

数。显示数组 array_1 的所有元素和数组元素的个数：

```
$ echo ${array_1[*]} ${#array_1[*]}
12 34 56 78 4
```

要给数组增加元素，可以直接定义。例如，给数组 y 增加第 3 个元素：

```
$ y[3]=kk
```

现在数组 y 应该有 4 个元素了。显示数组 y 的所有元素和数组元素的个数：

```
$ echo ${y[*]} ${#y[*]}
aa bb cc kk 4
```

要取消数组的定义，用命令 unset <数组名>，去除某个元素用命令 unset <数组名[下标]>。去除 y 的某个元素后，再看看数组 y 的所有元素及元素个数：

```
$ unset y[1]
$ echo ${y[*]} ${#y[*]}
aa cc kk 3
```

显示数组元素的值的时候，大括号是需要的，如果没有大括号，显示结果不对：

```
$ echo $y[3]
aa[3]                # 没有正常显示数组的元素值，用${y[3]}则正确
```

没有大括号时，$与 y 结合在一起，$y 表示数组的首个（第 0 个）元素，所以显示了 aa，并且后面的[3]原封不动地显示出来，因此最后显示 aa[3]。

还可以用带选项-a 的内置命令 declare 定义数组，格式为：

```
declare -a  数组名
```

用命令 declare 定义数组 w，再分别给数组元素赋值，数组下标可以不从 0 开始，也可以不连续：

```
$ declare -a w
$ w[1]=121
$ w[2]=abc
$ w[5]=835
```

显示数组 w 的元素个数和所有元素的值，但是看不到相应的数组下标：

```
$ echo ${#w[*]} ${w[*]}
3 121 abc 835
```

通过${!数组名[*]}可以得到数组的所有下标：

```
$ echo ${!w[*]}
1 2 5
```

通过${数组名[@]}也可以查看数组的所有元素：

```
$ echo ${w[@]}
121 abc 835
```

${数组名[@]}与${数组名[*]}，都可以用来查看数组的所有元素。但是它们有区别：echo ${w[*]}取出 w 的所有的元素作为一个字符串显示出来，echo ${w[@]} 取出每个元素，以空白间隔显示出来。在后面讲 for 循环的时候，有相应的例子，可以看出它们之间的区别。

用 declare 命令的选项-p 可以显示变量的属性和值。例如，下面命令的结果可以知道 w 的属性是-a，表示 w 是数组（array）；w 的值，也就是数组 w 的每个元素的下标和每个元素值，都很清楚地显示出来：

```
$ declare -p w
declare -a w='([1]="121" [2]="abc" [5]="125")'
```

定义数组 w 的元素值的时候，明明输入了 w[1]=121，w[5]=125，看上去是整数定义，为什么用命令 declare -p 显示数组的时候，121 和 125 两边有了双引号了呢？这是因为本质上 Bash 只有字符串类型，所以无论给数组元素赋什么值，Bash 都默认将它当作字符串处理。

还可以用([下标 1]=值 1 [下标 2]=值 2 …)的格式来定义数组，例如：

```
$ w=([2]=ok [5]=100 [8]=299 [10]=bye)
```

如果一个数组的下标不从 0 开始，或者数组下标不连续，可以用上面这种方法定义。查看一下数组 w，可知定义成功：

```
$ declare -p w
declare -a w='([2]="ok" [5]="100" [8]="299" [10]="bye")'
```

Bash 的内置命令 readarray 与 mapfile 的功能一致，是将标准输入的内容读入一个数组，每一行成为一个数组元素。

```
$ mapfile array1
28          # 键盘输入 4 个数，按〈Ctrl+D〉键结束
67
77
54
```

查看一下数组 array1 的内容：

```
$ declare -p array1
declare -a array1='([0]="28
" [1]="67
" [2]="77
" [3]="54
")'
```

array1 的内容没有在同一行显示，这是因为，键盘输入时，28、67、77 等后面的换行符被当作了元素值的一部分。使用选项-t 可以去掉行尾的换行符（trailing newline）：

```
$ mapfile -t array1
28          # 键盘输入 4 个数，按〈Ctrl+D〉键结束
67
77
54
```

这时，28、67、77 等后面的换行符在赋值时被去掉了，array1 的内容在同一行显示：

```
$ declare -p array1
declare -a array1='([0]="28" [1]="67" [2]="77" [3]="54")'
```

键盘输入数组元素的值容易出错，可以使用输入重定向。例如：

```
$ cat file.txt                        # 文件 file.txt 有 4 个数
28
67
77
54
$ mapfile -t array2 < file.txt        # 使用输入重定向定义数组 array2
$ declare -p array2                   # 查看一下数组 array2，可见定义成功
declare -a array2='([0]="28" [1]="67" [2]="77" [3]="54")'
```

mapfile 命令还可以使用文件描述符来定义数组，选项及参数为-u fd，fd 代表文件描述符。看下面的例子：

```
$ exec 12< file.txt                   # 将文件读入到文件描述符 12
$ mapfile -t -u 12 array3             # 将描述符 12 的内容读入到数组 array3 中
```

查看一下数组 array3，可见定义成功：

```
$ declare -p array3
declare -a array3='([0]="28" [1]="67" [2]="77" [3]="54")'
```

可以用命令 readonly -a <数组名>或者 declare -ar <数组名>使得一个数组成为只读数组。不能用 unset 命令取消只读数组的定义（不能取消整个数组的定义，不能取消某个元素的定义），也不能改变只读数组的定义（不能增加新元素，不能改变已有的元素）。接上面的例子，例如：

```
$ readonly -a array3
$ array3[4]=99
bash: array3: readonly variable
$ unset array3[0]
bash: unset: array3: cannot unset: readonly variable
$ unset array3
bash: unset: array3: cannot unset: readonly variable
```

4.10　关联数组

关于数组的下标，人们通常会想它应该是正整数或非负整数，这也容易理解和接受。实际上，Bash4 以上的版本，像 Perl 语言一样，数组的下标可以不是整数了。这样的数组叫做关联数组（Associative Arrays），也称为哈希（hash）或映射（mapping）。运行 help declare，查看内置命令 declare 的帮助，如果能看到选项-A，就说明正在使用的 Bash 支持关联数组。上一节讲的数组叫索引数组（Indexed Arrays）。

如果用数组记录人的年龄，或者记录水果的价格，可以将人或者水果的编号作为数组下标，但还需要存储人名（水果名）与人的编号（水果编号）的对应关系。如果把人名或者水果名直接作为数组下标，是不是很方便？通过例子，可以知道其方便之处，用 declare 加选项-A 定义如下两个关联数组：

```
$ declare -A age=([Mike]=28 [Jack]=35 [Tom]=19)
$ declare -A price=([apple]=5.5 [banana]=2.3 [orange]=1.5)
```

定义了关联数组 age，它记录了 Mike、Jack 和 Tom 的年龄，分别是 28、35 和 19；定义了关联数组 price，它记录了 apple、banana 和 orange 的价格，分别是 5.5、2.3 和 1.5。查看一下，定义成功：

```
$ declare -p age
declare -A age='([Jack]="35" [Tom]="19" [Mike]="28" )'
$ declare -p price
declare -A price='([orange]="1.5" [apple]="5.5" [banana]="2.3" )'
```

从上面的结果可见，显示出来的元素的顺序和定义时的顺序是不同的，因为它们是用散列法存储，而不是像普通数组那样使用顺序存储。

这时，可以查询 Tom 的年龄和 apple 的价格：

```
$ echo ${age[Tom]}
19
$ echo ${price[apple]}
5.5
```

定义了关联数组，再查询某人的年龄，或者某种水果的价格，比普通数组更方便、更明了。关联数组 age 当中的人名 Jack、Tom 和 Mike 不叫数组下标了，而叫键（key）；35、19 和 28 是对应的值（value）。查看关联数组 age 的全部的键和相应的值：

```
$ echo ${!age[@]}
Jack Tom Mike
$ echo ${age[@]}
35 19 28
```

同样可以用 readonly -A 加关联数组名或者 declare -Ar 加关联数组名使它成为只读关联数组，不能用 unset 命令取消只读关联数组的定义，也不能改变只读关联数组的定义。

命令 declare -Ap 可以显示当前 shell 的所有的关联数组，命令 declare -ap 显示所有的普通数组（索引数组）。

4.11　导出变量命令 export 和 declare -x

给一个变量赋值后，该变量只是在当前 shell 起作用，进入子 shell 后该变量就不起作用了。例如：

```
$ full_name="Michael Scofield"
$ echo $full_name
Michael Scofield
$ bash                # 执行 bash 进入子 shell
$ echo $full_name
                      # 显示空行
$ exit                # 退出 shell，返回原来的父 shell
$ echo $full_name
Michael Scofield
```

上面的例子，给字符串变量 full_name 赋值，显示出了其内容。执行 bash 后，就进入子 shell，子 shell 中的变量 full_name 没有定义。执行 exit，回到父 shell，就是回到原来的 shell，再执行 echo $full_name，又可以得到预期的显示结果。

内置命令 export 可以使子 shell 继承父 shell 的变量定义。下面先定义变量，再将变量导出（export），进入子 shell 后，再查看变量的值，可见子 shell 继承了父 shell 的变量定义：

```
$ full_name="Michael Scofield"
$ export full_name          # 导出变量
$ bash                      # 执行 bash 进入子 shell
$ echo $full_name
Michael Scofield            # 子 shell 继承了父 shell 的变量 full_name
```

上面例子的前两条命令可以合为一条，export <变量名>=<值>，即：

```
$ export full_name="Michael Scofield"
```

在 shell terminal 里运行一个脚本时，脚本当中的命令是在一个子 shell 里面运行，它与父 shell 不是同一个 shell，见图 3-4。

再来理解一下图 3-4。看个例子，脚本 display_age.sh 用来显示 age 的值：

```
$ cat display_age.sh
#!/bin/bash
echo "age=$age"
```

下面定义 age 为 25，再执行脚本:

```
$ age=25
$ display_age.sh
age=
```

age 的值没有显示出来。因为 age=25 是在当前 shell 里面定义的，而脚本 display_age.sh 中的命令 echo "age=$age"是在一个子 shell 里面运行，当前 shell 的 age 并不影响其子 shell 的 age。也可以说：子 shell 的 age 和当前 shell 的 age 不是同一个变量，脚本 display_age.sh 中并未对 age 赋值，所以显示 age 的值为空。在 age=25 的前面加上 export，再执行脚本：

```
$ export age=25
$ display_age.sh
age=25
```

这时，当前 shell 的变量 age 被导出，对子 shell 可见，脚本 display_age.sh 中的 echo 命令就可以显示 age 的值了。

export 可以使变量的定义“穿透”多层 shell。它不仅使父 shell 的变量定义导入到子 shell，还可以导入到子 shell 的子 shell，可以一直继承下去，如果有多层子 shell 的话。

运行命令 declare -x <变量名>也可以导出变量，效果与命令 export <变量名>相同。

就像 set 的逆命令是 unset，export 的逆命令是 export -n，选项-n 的作用是取消变量的导出。取消前面提到的变量 age 的导出，再运行脚本 display_age.sh，就又像最开始一样，显示 age 的值为空：

```
$ export -n age
$ display_age.sh
age=
```

命令 export -p 和 declare -xp 的作用是显示当前 shell 中所有已经导出的变量。例如，在某 Linux 系统中，运行 export -p，部分内容如下：

```
$ export -p
declare -x COLORTERM="gnome-terminal"
declare -x DEFAULTS_PATH="/usr/share/gconf/gnome-2d.default.path"
declare -x DESKTOP_SESSION="gnome-2d"
declare -x DISPLAY=":0.0"
declare -x GDMSESSION="gnome-2d"
declare -x GDM_KEYBOARD_LAYOUT="us"
declare -x GDM_LANG="en_US.utf8"
declare -x GNOME_DESKTOP_SESSION_ID="this-is-deprecated"
declare -x GNOME_KEYRING_CONTROL="/tmp/keyring-fGmUUY"
declare -x GTK_MODULES="canberra-gtk-module"
declare -x HOME="/home/user"
```

4.12　declare 命令总结

命令 declare 用来定义变量及其属性，还可设置函数属性（见第 7 章），格式如下：

```
declare [-aAfFgilrtux] [-p] [name[=value] ...]
```

选项作用为：

-f，显示函数名和函数定义，或限定对函数（而不是变量）操作
-F，仅显示函数名
-g，在函数中，用来定义全局变量
-p，显示属性及值

其中，用来设定属性的选项有：

-a，定义索引数组
-A，定义关联数组
-i，定义整型数（使变量有整数属性）
-l，赋值时将字母转为小写（如果有大写字母）
-r，赋予只读属性
-t，赋予 trace 属性（对变量无效，对函数有效）
-u，赋值时将字母转为大写（如果有小写字母）
-x，导出变量或者函数

例如，要保证变量赋值时，如有字母，得到的全是小（大）写字母，则可用选项-l（-u）：

```
$ declare -l x="This IS A Book"
$ echo $x
this is a book
$ declare -u x="This IS A Book"
$ echo $x
THIS IS A BOOK
```

4.13　环境变量与特殊变量

前面已经提到过环境变量，例如 PS1 和 PATH 等，环境变量名一般使用大写字母。运行 env 或者 printenv 命令可以显示环境变量，例如：

```
$ printenv
ORBIT_SOCKETDIR=/tmp/orbit-user
SSH_AGENT_PID=1490
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=53e504fc8e5e27454d11d74800000015-1346896806.796690-1045857240
WINDOWID=54525982
GNOME_KEYRING_CONTROL=/tmp/keyring-X9tToI
```

```
GTK_MODULES=canberra-gtk-module
USER=user
SSH_AUTH_SOCK=/tmp/keyring-X9tToI/ssh
SESSION_MANAGER=local/ubuntu:@/tmp/.ICE-unix/1457,unix/ubuntu:/tmp/.ICE-unix/1457
USERNAME=user
DEFAULTS_PATH=/usr/share/gconf/gnome-2d.default.path
XDG_CONFIG_DIRS=/etc/xdg/xdg-gnome-2d:/etc/xdg
PATH=/home/user/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
DESKTOP_SESSION=gnome-2d
PWD=/home/user
GDM_KEYBOARD_LAYOUT=us
LANG=en_US.UTF-8
GDM_LANG=en_US.utf8
MANDATORY_PATH=/usr/share/gconf/gnome-2d.mandatory.path
UBUNTU_MENUPROXY=libappmenu.so
GDMSESSION=gnome-2d
SHLVL=1
HOME=/home/user
LANGUAGE=en_US:en
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=user
XDG_DATA_DIRS=/usr/share/gnome-2d:/usr/share/gnome:/usr/local/share/:/usr/share/
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-pewKAVUbBG,guid=a87aa092e51a08
5b27404fed0000002e
LESSOPEN=| /usr/bin/lesspipe %s
WINDOWPATH=7
DISPLAY=:0.0
LESSCLOSE=/usr/bin/lesspipe %s %s
COLORTERM=gnome-terminal
XAUTHORITY=/var/run/gdm/auth-for-user-QUQMbp/database
```

上面输出的环境变量列表没有按照字母表排序，并且 PS1 这个环境变量也不在其列。env 与 printenv 都不是 Bash 的内置命令，运行 Bash 的内置命令 set，可以看见当前 shell 的所有的按照字母排序的普通变量、数组、环境变量和函数定义等。命令 set 的输出较长，运行 set | more 可以分页查看。

不同的计算机，或同一计算机的不同的账户，环境变量都不完全一样。管理员可以设置公共的环境变量，用户可以设置自己账户的环境变量。

在 Linux 中定义变量的习惯是：普通变量用小写，环境变量用大写。定义普通变量和环境变量的格式是一样的，都是“变量名=值”。多数情况下，普通变量不需要导出、而环境变量需要导出，所以，也可以认为定义环境变量的格式为：

```
export 环境变量名=值
```

举一个环境变量应用的例子，也是接着 2.5 节的内容继续讲解。如果 code 目录放在/tmp 目录下，可设置环境变量 export CODE_DIR=/tmp；如果放在/home/user 目录下，可设置

export CODE_DIR=/home/user；如果放在当前账户的主目录下，运行 export CODE_DIR=$HOME。那么，脚本 A 调用脚本 C 就用${CODE_DIR}/code/driver/cust/C，脚本 A 调用脚本 B 就用${CODE_DIR}/code/config/B。运行脚本 A 之前，把环境变量 CODE_DIR 的值设置正确即可，无需把脚本 A 的内容改来改去了。

Bash 自带了一些特殊的内置变量，简称为 Bash 特殊变量，它们是后面要讲的 Bash 内置变量的一部分，见表 4-2。

表 4-2 Bash 的特殊变量

变量	含 义
$0	脚本自身的名字，或者 shell 的名字（如 bash，tcsh）
$N	脚本或函数的位置参数$1，$2，…，${10}，…，N 大于 9 时需用大括号括起来
$#	位置参数个数
$*	所有的位置参数（一起作为单个字符串）
$@	所有的位置参数（每个都作为独立的字符串）
$?	上一条命令的退出状态值
$$	当前 shell 的进程 ID
$!	最后一个后台命令的进程 ID
$-	当前 shell 的选项（见第 10 章）
$_	上一条命令的最后一个参数

下面举例讲解几个比较常用的 Bash 的特殊变量。在某个脚本里包含$0 时，$0 的值就是脚本名字本身：

```
$ cat env_dollar_0.sh
#!/bin/bash
echo This script name is $0
```

运行脚本 env_dollar_0.sh：

```
$ env_dollar_0.sh
This script name is ./env_dollar_0.sh
```

对于某个脚本而言，$1，$2，$3 分别表示脚本的前 3 个参数，$#表示参数个数，$*和$@都表示所有的参数（但$*和$@不完全一样）。

阅读脚本 positional_parameter_1.sh 和其运行结果，可以理解表 4-2 中的$#，$N（N=1,2,3,…），$*和$@的作用。

```
$ cat positional_parameter_1.sh
#!/bin/bash
echo There are $# parameters
echo Parameters are $1, $2, $3.
echo All parameters are $*
echo All parameters are $@
```

分别输入 1 个、3 个和 5 个参数，看看其运行结果：

```
$ positional_parameter_1.sh par1                    # 脚本有 1 个参数 par1
There are 1 parameters
Parameters are par1, , .                            # $2 与$3 的值为空
All parameters are par1
All parameters are par1
$ positional_parameter_1.sh par1 par2 par3          # 脚本有 3 个参数 par1, par2, par3
There are 3 parameters
Parameters are par1, par2, par3.
All parameters are par1 par2 par3
All parameters are par1 par2 par3
$ positional_parameter_1.sh par1 par2 par3 par4 par5    # 脚本有 5 个参数 par1 到 par5
There are 5 parameters
Parameters are par1, par2, par3.                    # 脚本中没写$4 和$5，最后两个未显示
All parameters are par1 par2 par3 par4 par5
All parameters are par1 par2 par3 par4 par5
```

从上面的例子看，似乎$*与$@的作用是相同的，其实不然，在讲 for 循环的时候有相应的例子说明它们的区别。

set 命令可以丢弃原来的位置参数并重置位置参数，set --可以清除所有的位置参数。$0 不受影响，它始终代表脚本名。看下面的脚本：

```
$ cat positional_parameter_2.sh
#!/bin/bash
echo There are $# parameters
echo All parameters are $*
set Jack Vickie Sam Mike Frank         # 重置位置参数，将 5 个人名设为脚本参数
echo There are $# parameters
echo All parameters are $*
set --                                 # 清除所有的参数
echo There are $# parameters
echo All parameters are $*
echo This script name is $0
```

下面执行脚本 positional_parameter_2.sh，给它传递 3 个参数 p1，p2，p3：

```
$ positional_parameter_2.sh p1 p2 p3
There are 3 parameters
All parameters are p1 p2 p3
There are 5 parameters                      # 执行了 set Jack … Frank，参数为 5 个
All parameters are Jack Vickie Sam Mike Frank
There are 0 parameters                      # 执行 set --之后，参数被清空
All parameters are                          # 这时，$*为空
This script name is ./positional_parameter_2.sh    # $0 始终代表脚本名
```

特殊变量$$表示进程 ID。想知道当前 shell 的进程 ID，运行 echo $$即可：

```
$ echo $$
1762
```

在某个 shell（称为父 shell）里面运行一个脚本，脚本的进程 ID 与其父 shell 的是不一样的。每次运行同一个脚本，脚本的进程 ID 都不一样，这是因为脚本里面的命令每次运行在不同的子 shell 里。例如，脚本 env_process_id.sh 显示$$的值，每次运行，值都不一样：

```
$ cat env_process_id.sh
#!/bin/bash
echo Process ID is $$
$ env_process_id.sh
Process ID is 5479      # 第一次运行，进程 ID 是 5479
$ env_process_id.sh
Process ID is 5495      # 中间停留一小段时间，再运行，进程 ID 是 5495
```

这里顺便讲解一下两种执行脚本方式的区别。一种方式是直接运行一个可执行脚本，每次运行都会产生一个新子 shell，每次会有不同的进程 ID，刚才的例子已经说明了这一点；另一种方式是运行 source <脚本名>（或者. <脚本名>），这种方式不产生新子 shell，而是在当前 shell 运行脚本中的命令。例如，运行 source env_process_id.sh，显示的就是当前 shell 的进程 ID：

```
$ source env_process_id.sh
Process ID is 1762
```

运行普通脚本，一般采用方式一，而且最好采用方式一，否则可能会破坏当前 shell 的环境。若要配置当前 shell 的环境（开发环境、编译环境），即运行一个为当前 shell 设置环境的脚本的时候，采用方式二。

表 4-2 中的$-表示当前 shell 的选项，很多选项可以用 set 命令设置或查询，见第 10 章。例如，运行 echo $-，如果$-含 i 表示当前 shell 为交互（interactive）shell；打开选项 noclobber 后，$-就包含了 C：

```
$ echo $-
himBH
$ set -o noclobber      # 打开选项 noclobber，用命令 set -C 也可以，见 3.9.3 节
$ echo $-
himBCH
```

表 4-2 中的$?在 3.7 节已经介绍过了。表 4-2 还包括其他特殊变量，这里不一一介绍了。

4.14 内置变量

表 4-3 列出了一些常用的 Bash 内置变量。

表 4-3　常用的 Bash 内置变量

变　　量	含义与作用
BASH	bash 的完整路径，默认为/bin/bash
BASH_ENV	仅在非交互模式中适用。在执行 shell 脚本时，会先检查该变量是否指定了启动脚本，若指定则先执行它指定的启动脚本
BASH_VERSION	Bash 的版本号
CDPATH	命令 cd 的搜索路径，多个路径用冒号（:）隔开
COLUMNS	用在内置命令 select 当中，用来设定输出选择列表时的终端宽度，自动根据 SIGWINCH 信号来设置
DIRSTACK	当前目录栈存放的数组
EUID	有效的当前用户的 ID
FUNCNAME	当某函数被调用时，该变量为函数名；实际上它是数组，记录调用链上所有的函数名
GLOBIGNORE	一个冒号分隔的模式列表，定义了文件名扩展时要忽略的文件名集合
HISTFILE	存放命令历史的文件，通常为~/.bash_history
HISTFILESIZE	命令历史文件可包含的最大行数
HISTIGNORE	不存入命令历史文件的指令序列，多个指令用冒号（:）隔开
HISTSIZE	可以保存的历史命令条数
HOME	用户的主目录
HOSTNAME	主机名
HOSTTYPE	主机的类型
IFS	字段分隔符，默认为空白符（空格，Tab 键，换行符）
INPUTRC	设定命令行函数或库 readline 的启动配置文件,可覆盖~/.inputrc 的设定
LANG	当前语系的名称
LC_ALL	设置当前的 locale,可覆盖 LANG 和 LC_*的设置
LC_CTYPE	设置 locale 的字符分类
LINENO	脚本中当前行号
LINES	用在内置命令 select 当中，控制菜单在终端显示垂直方向上的行数
LOGNAME	当前用户的登录名
MACHTYPE	描述主机形态的 GNU 格式：CPU-公司-系统，如 i686-pc-linux-gnu
MAIL	用来通知用户有邮件到达
MAILCHECK	多长时间检查一次新邮件，单位为秒，默认为 60 秒
OLDPWD	前一个工作目录，命令 cd -等价于命令 cd $OLDPWD
OPTARG	存放内置命令 getopts 参数的值
OPTIND	待处理的 getopts 下一个参数的索引，初始值为 1
PATH	外部命令的搜索路径，多个路径以冒号（:）隔开
PPID	父进程（父 shell）的进程 ID
PS1	命令行主提示符，默认为“\s-\v\$”
PS2	命令续行提示符，默认为“>”
PS3	命令 select 的提示符，默认为“#?”
PS4	命令 set 的-x 选项启用后，在命令行前的提示符，默认为“+”
PWD	当前的工作目录，命令 echo $PWD 等价于命令 pwd
RANDOM	0 到 32767 之间的一个随机数

（续）

变　　量	含义与作用
REPLY	read 命令后不跟变量名时，键盘输入存入 REPLY；select 命令，用户的选择项存入 REPLY
SECONDS	当前 shell 的启动的时间，单位为 s
SHELL	登录 Linux 后的默认 shell，查看/etc/shells 可知允许使用的 shell，运行命令 chsh 可以设置，/bin/bash 与/bin/tcsh 比较常用
SHELLOPTS	shell 的选项，选项之间以冒号（:）间隔
SHLVL	第一次打开一个 shell 终端，它的值为 1，每进入一层子 shell，它的值增加 1
TMOUT	如果该变量的值大于 0，当前 shell 在等待 TMOUT 秒之后没有任何输入就会自动退出，即超时（time out）退出
UID	用户 ID 号

下面举例说明表 4-3 中的一部分内置变量的含义与使用。

例如，查看内置变量 BASH 和 BASH_VERSION 的值：

```
$ echo $BASH
/bin/bash
$ echo $BASH_VERSION
4.2.8(1)-release
```

查看当前 shell 的启动时间，即 shell 窗口从打开到现在已经过去秒数：

```
$ echo $SECONDS
8743
```

下面的例子用来说明内置变量 CDPATH 的作用。建立目录 A，B 和 C，并保证子目录 C 的父目录是/tmp/A/B：

```
$ mkdir -p /tmp/A/B/C
```

在/home/user 目录下，第一次运行 cd C 是失败的，因为在/home/user 目录下没有子目录 C：

```
$ pwd
/home/user
$ cd C
bash: cd: C: No such file or directory        # 在/home/user 目录下没有子目录 C
```

第二次运行 cd C 之前，定义了内置变量 CDPATH，将目录/tmp/A/B 加入 cd 命令的搜索路径 CDPATH：

```
$ CDPATH="/tmp/A/B:/usr/bin"
```

这时运行 cd C 时，系统会在内置变量 CDPATH 包含的路径中查找，并在/tmp/A/B 目录下发现 C 目录。所以在/home/user 目录下直接运行 cd C，就可以进入/tmp/A/B/C 目录，而不需要运行 cd /tmp/A/B/C，即不需要输入全路径：

```
$ cd C
/tmp/A/B/C                  # 显示/tmp/A/B/C，表明进入到了/tmp/A/B/C 目录
```

再用 pwd 命令确认一下，确实进入了/tmp/A/B/C 目录：

```
$ pwd
/tmp/A/B/C
```

内置命令 history 可以列出历史命令，内置变量 HISTFILE（history file 的意思）用来指定存放历史命令的文件，通常 HISTFILE=~/.bash_history。在用户退出 shell 时，历史命令会存放在主目录下面的.bash_history 文件中。

内置变量 HISTIGNORE（history ignore 的意思）用来指定不希望被记录在历史命令列表中的命令或者命令序列，多个命令用冒号隔开。例如，设置 HISTIGNORE=ls:t*:\&，那么命令 ls 和以 t 开头的指令不会被记录在历史命令列表中，&表示上一条命令，&可以使得连续输入的相同的命令只被记录一次，避免重复记录。因为在命令行&表示将命令放在后台，所以在&的前面加反斜杠屏蔽其特殊含义。下面看具体的例子。history 命令的选项-c 用来清空（clear）历史命令列表。为便于看清例子，先清空历史命令列表：

```
$ history -c
```

设置在命令的历史记录中将被忽略的命令序列：

```
$ HISTIGNORE=ls:t*:\&
$ echo $HISTIGNORE
ls:t*:&
```

下面分别运行 date，ls，pwd（重复三次），touch 命令：

```
$ date
Mon Oct 15 10:42:16 EDT 2012
$ ls
[文件清单…]
$ pwd
/home/user
$ pwd
/home/user
$ pwd
/home/user
$ touch /tmp/email_draft
```

现在，运行 history 命令，得到的历史命令列表不包含 ls 和 touch，并且 pwd 只出现一次。因为 HISTIGNORE 包含 ls 和 t*，所以 ls 和 touch 不被记录在历史命令列表中；因为 HISTIGNORE 包含&，当刚刚运行完第二个和第三个 pwd 命令时，系统发现其与上一条命令完全相同，所以系统不将其记录，只记录第一个 pwd：

```
$ history
    1  HISTIGNORE=ls:t*:\&
    2  echo $HISTIGNORE
    3  date
    4  pwd
    5  history
```

内置环境变量 HOME 记录着用户的主目录。账户 user 默认的主目录为/home/user，运行 cd 或者 cd ~就进入用户的主目录。一般地，HOME 的值无需改动，也最好不要改动。如果改变 HOME 的值，将影响使用它的命令和脚本的行为，至少影响命令 cd 和 cd ~的运行结果。见下例：

```
$ cd
$ pwd
/home/user                    # 运行命令 cd，进入了默认主目录
$ mkdir book
$ HOME=/home/user/book        # 现在将主目录改为/home/user/book
$ cd
$ pwd
/home/user/book               # 运行命令 cd，进入了新的主目录
```

内置环境变量 PWD 的值是当前的工作目录，echo $PWD 等价于命令 pwd，例如：

```
$ pwd
/home/user
$ echo $PWD
/home/user
```

假设在/home/user 目录里面有一个脚本 current_directory.sh，它的第一条命令显示当前的工作目录，接下来的两条命令创建目录/tmp/maggie/bin 并进入其中，最后一条命令显示执行 cd /tmp/maggie/bin 之后的工作目录：

```
$ cat current_directory.sh
#!/bin/bash
echo $PWD
mkdir -p /tmp/maggie/bin; cd /tmp/maggie/bin
echo $PWD
```

执行脚本 current_directory.sh：

```
$ current_directory.sh
/home/user            # 脚本第一条命令显示当前的工作目录/home/user
/tmp/maggie/bin       # 进入/tmp/maggie/bin 之后的工作目录
```

现在出现一个问题，脚本 current_directory.sh 运行完成了，那么当前的工作目录是运行脚本之前的目录/home/user 呢，还是脚本的最后一条命令所显示的目录/tmp/maggie/bin 呢？试试就知道了：

```
$ echo $PWD
/home/user
```

可见，当前的工作目录是运行脚本之前的工作目录。这说明，脚本运行时所在的子 shell 的工作目录的改变不影响父 shell 的工作目录（其实在 3.11.8 节已经讲过：子 shell 中的路径变化不影响父 shell）。如果用 source 运行脚本，则脚本就在当前 shell（而不是在一个子

shell）运行，那么当前 shell 的工作目录会发生改变：

```
$ source current_directory.sh
/home/user
/tmp/maggie/bin
$ pwd                       # 或者运行 echo $PWD
/tmp/maggie/bin
```

内置环境变量 OLDPWD 记录的是前一次的工作目录，命令 cd -的作用是回到上一次的工作目录，与 cd $OLDPWD 作用一致。见下例：

```
$ pwd
/home/user                  # 当前目录为/home/user
$ cd book
$ pwd
/home/user/book             # 当前目录为/home/user/book
$ cd ..
$ pwd
/home/user                  # 运行 cd ..后，回到主目录/home/user
$ echo $OLDPWD
/home/user/book             # OLDPWD 记录的是前一次的工作目录/home/user/book
$ cd $OLDPWD                # 这里用命令 cd -，效果是一样的
$ pwd
/home/user/book             # 运行 cd $OLDPWD 后，进入了刚才的工作目录/home/user/book
```

内置环境变量 TMOUT 默认是没有值的。若该变量的值大于 0，则在等待 TMOUT 秒后还没有任何输入时，当前的 shell 就会自动退出。例如，设置它为 20：

```
$ TMOUT=20
```

那么在 20s 之内，没有任何键盘输入的话，当前 shell terminal 将自动关闭。

以上列举了表 4-3 中的部分内置变量，还有很多其他的内置变量，这里不一一讲解。

4.15　计算表达式值的命令 expr

外部命令 expr 用于计算表达式的值，格式为：

```
expr 表达式
```

例如：

```
$ expr 1 + 2
3
```

expr 表达式中的运算符两边都需要有空格，否则运算无法进行：

```
$ expr 1+2
```

```
1+2          # 加号两边没有空格，所以没有得到 3
$ expr 12 % 5
2            # 12 除以 5 的余数为 2
$ expr 30 / 3
10
```

遵循先乘除，后加减的规则：

```
$ expr 5 + 30 / 5
11
```

进行乘法的时候要注意，因为*在 Linux 命令里面有特殊的意图，*将被替换为当前目录下面所有的文件，所以下面的命令出错：

```
$ expr 8 * 6
expr: syntax error
```

如果恰好在一个空目录下面运行 expr 8 * 6，是可以成功的。可以创建一个新目录，进入新目录再运行 expr 8 * 6 试试。

用反斜线屏蔽*的特定含义，就可以正常进行乘法运算：

```
$ expr 8 \* 6
48
```

用命令 expr 可以进行变量参与的运算：

```
$ i=3
$ expr $i + 5
8
```

用命令 expr 可以进行变量自增运算，常用在循环当中：

```
$ i=1
$ i=$(expr $i + 1)
$ echo $i
2
$ i=`expr $i + 1`
$ echo $i
3
```

使用 expr 进行计算的时候，变量必须是整数，不能是字符串，也不能含小数，否则会出错（命令的退出状态为非 0）：

```
$ i=hello
$ expr $i + 58
expr: non-integer argument
$ echo $?
2
```

利用命令 expr 的这一特点，可以判断某个变量是不是整数。让其参与整数运算，比如，与 1 相加，如果 expr 命令的退出状态是 0（成功），表明该变量是整数，否则不是整数：

```
$ k=5
$ expr 1 + $k > /dev/null 2>&1           # 这里只关心命令的退出状态，不关心值是多少，
                                          # 所以将标准输出与标准错误都重定向到“黑洞”
$ echo $?
0                                         # 退出状态为 0，说明 k 是整数
$ m=5.8
$ expr 1 + $m > /dev/null 2>&1
$ echo $?
2                                         # 退出状态非 0，说明 m 不是整数
```

expr 还可以应用在关系运算中，如，<、<=、!=、=、>、>=等。前面提到过，与 C 语言一样，关系运算结果，0 代表假，1 代表真：

```
$ i=6 j=8
$ expr $i = $j
0    # 6 等于 8 是假的
$ expr $i != $j
1    # 6 不等于 8 是真的
```

因为>与<在 Linux 的命令行里代表输出重定向和输入重定向，所以在使用命令 expr 进行大小关系运算（比较）时，需要用反斜线屏蔽其特定含义：

```
$ expr $i >= $j
expr: syntax error
```

用反斜线屏蔽>，因为 6>=8 是不成立的，所以下面的关系运算结果是 0（假）：

```
$ expr $i \>= $j
0
```

用反斜线屏蔽<，因为 6<8 是成立的，所以下面的关系运算结果是 1（真）：

```
$ expr $i \< $j
1
```

4.16　变量测试及其相应的赋值

在编写脚本时，在一个变量未赋初值的情况下，有时需要给它一个默认值。Bash 提供了这方面的功能，见表 4-4。

表 4-4　变量替换

表 达 式	含　　义
${var-Default}	如果未定义 var，那么就以 Default 作为表达式的值，var 不变
${var:-Default}	如果未定义 var，或者定义了 var 但值为空，那么就以 Default 作为表达式的值，var 不变
${var=Default}	如果未定义 var，那么就以 Default 作为表达式的值，同时赋值 var 为 Default
${var:=Default}	如果未定义 var，或者定义了 var 但值为空，那么就以 Default 作为表达式的值，同时赋值 var 为 Default
${var+Other}	如果定义了 var（有值或者赋空值），那么表达式的值就是 Other，var 不变
${var:+Other}	如果定义了 var（非空），那么表达式的值就是 Other，var 不变
${var?Message}	如果未定义 var，那么就打印 Message，var 不变
${var:?Message}	如果未定义 var，或者定义了 var 但值为空，那么就打印 Message，var 不变
${!varprefix*}	匹配之前所有声明的以 varprefix 开头的变量
${!varprefix@}	匹配之前所有声明的以 varprefix 开头的变量

先定义 var，保证 var 非空，表达式${var-Default}或者${var:-Default}的值就是 var 的值，例如：

```
$ var=20
$ echo ${var:-8}
20
```

var 本身不受影响：

```
$ echo $var
20
```

如果 var 未定义，表达式${var-Default}和${var:-Default}的值就是 Default，var 本身不受影响，例如：

```
$ unset var     # 先取消它的定义
$ echo ${var:-8}
8
$ echo ${var-8}
8
```

如果 var 定义了，但值为空，表达式${var-Default}的值为空，而表达式${var:-Default}的值就是 Default，var 本身不受影响，例如：

```
$ var=              # 等号后面直接回车，给它赋空值
$ echo ${var-8}
                    # 显示空行
$ echo ${var:-8}
8
```

对于用户而言，一个变量未定义和定义为空，二者效果是一样的，都是没有值，所以${var:-Default}比较常用，${var-Default}不太常用。同样道理，${var:=Default}比较常用。下

面只讲${var:=Default}的使用。

如果 var 有值，表达式${var:=Default}的值就是 var 的值，例如：

```
$ str=a_string
$ echo ${str:=test_string}
a_string
```

如果 var 无值，表达式${var:=Default}的值就是 Default，同时 var 被赋值为 Default（如果 var 无值，表达式${var:-Default}的值也是 Default，但 var 不变，这是${var:=Default}与${var:-Default}的区别所在）。例如：

```
$ unset str                    # 先取消 str 的定义，确保它无值
$ echo ${str:=test_string}
test_string                    # 表达式的值为 test_string
$ echo $str
test_string                    # 同时变量 str 被赋值，值也为 test_string
```

实际上，表 4-4 中的 Default 也可以是一个变量，例如：

```
$ str="morning"
$ test_str="afternoon"
$ echo ${str:=$test_str}
morning
$ unset str          # 取消 str 的定义，确保它无值
$ echo ${str:=$test_str}
afternoon            # 表达式的值为变量 test_str 的值 afternoon
$ echo $str
afternoon            # 变量 str 被赋值为 test_str 的值 afternoon
```

下面的脚本询问客户的需求，客户需要什么，就显示客户的需求，客户无明确需求的，默认为客户需要水。

```
$ cat need.sh
#!/bin/bash
echo "What do you need?"
read need
echo "I need ${need:-water}."
```

运行脚本，输入 beer，显示客户需要 beer：

```
$ need.sh
What do you need?
beer                           # 输入 beer
I need beer.
```

运行脚本，不输入需求（直接按回车键），显示客户需要 water：

```
$ need.sh
```

```
What do you need?
[按回车键]                    # 不输入，直接按回车键
I need water.
```

+与-是相反的运算，${var+Other}或${var:+Other}与${var-Default}或${var:-Default}的作用刚好相反。当 var 有值的时候，${var+Other}与${var:+Other}的值都是 Other：

```
$ var=99
$ echo ${var+88}
88
$ echo ${var:+88}
88
```

var 本身不受影响：

```
$ echo $var
99
```

当 var 未定义时，${var+Other}与${var:+Other}的值都是空：

```
$ unset var    # 先取消 var 的定义，确保它未定义
$ echo ${var:+88}
               # 空行
$ echo ${var+88}
               # 空行
```

当 var 有定义但为空时，${var+Other}的值为 Other，${var:+Other}的值为空：

```
$ var=         # 等号后面直接回车，给它赋空值
$ echo ${var+88}
88
$ echo ${var:+88}
               # 空行
```

如果 var 有非空值，${var?Message}与${var:?Message}的值就是 var 本身，例如：

```
$ var="study shell"
$ echo ${var?"it is not defined"}
study shell
$ echo ${var:?"it is not defined"}
study shell
```

如果 var 未定义，${var?Message}与${var:?Message}会显示 Message，例如：

```
$ unset var     # 先取消 var 的定义，确保它未定义
$ echo ${var?"it is not defined"}
bash: var: it is not defined
$ echo ${var:?"it is not defined"}
```

```
bash: var: it is not defined
```

当 var 未定义时，在命令行直接输入${var?Message}或者${var:?Message}就会显示 Message：

```
$ ${var?"it is not defined"}
bash: var: it is not defined
$ ${var:?"it is not defined"}
bash: var: it is not defined
```

如果 var 被赋为空，${var?Message}显示空，${var:?Message}会显示 Message，例如：

```
$ var=          # 等号后面直接按回车键，给它赋空值
$ echo ${var?"it is not defined or it is null"}
                # 空行
$ echo ${var:?"it is not defined or it is null"}
bash: var: it is not defined or it is null
```

下面举一个${var:?Message}应用的例子。脚本 positional_parameter_3.sh 的第二行先检查位置参数$1 是否有值，如果有，把值赋给 name，然后第三行显示 Good morning；如果位置参数$1 没有值，第二行显示提示 please input a name。

```
$ cat positional_parameter_3.sh
#!/bin/bash
name=${1:?"please input a name"}
echo "Good morning $name"
```

运行脚本，参数为 Tom：

```
$ positional_parameter_3.sh Tom
Good morning Tom
```

无参数，运行脚本：

```
$ positional_parameter_3.sh
./positional_parameter_3.sh: line 2: 1: please input a name
```

当运行 positional_parameter_3.sh Tom 时，参数为 Tom，即位置参数$1=Tom，那么表达式${1:?"please input a name"}的值为 Tom，则变量 name 被赋值为 Tom，最后显示 Good morning Tom。当运行 positional_parameter_3.sh 时，没有参数，即位置参数$1 无值，执行到第二行时，显示 please input a name，第三行不再执行。

下面介绍表 4-4 中${!varprefix*}的使用。

定义三个以字母 v 开头的变量：

```
$ v1=11
$ v2=12
$ v3=13
```

列出当前 shell 以 v 开头的变量：

```
$ echo ${!v*}
v1 v2 v3
```

再定义两个值为空的以 v 开头的变量：

```
$ v4=                    # 等号后面直接回车
$ v5=
```

列出当前 shell 以 v 开头的变量：

```
$ echo ${!v*}
v1 v2 v3 v4 v5
```

4.17　字符串操作

4.17.1　Bash 内置的字符串操作

Linux 下的常用工具 awk 与 sed 可以做与字符串相关的很多操作。Bash 内置了一些字符串的操作功能，使用内置操作可以省掉启动外部程序的时间，速度会更快，见表 4-5。

表 4-5　字符串操作

表　达　式	含　　义
${#string}	字符串 string 的长度
${string:position}	从 string 中位置 position 开始提取子串
${string:position:length}	从 string 中位置 position 开始提取长度为 length 的子串
${string#regexp}	从变量 string 的开头，删除最短匹配 regexp 的子串
${string##regexp}	从变量 string 的开头，删除最长匹配 regexp 的子串
${string%regexp}	从变量 string 的结尾，删除最短匹配 regexp 的子串
${string%%regexp}	从变量 string 的结尾，删除最长匹配 regexp 的子串
${string/regexp/replacement}	使用 replacement，来代替第一个匹配的 regexp
${string//regexp/replacement}	使用 replacement，代替所有匹配的 regexp
${string/#regexp/replacement}	如果 string 的前缀匹配 regexp，那么就用 replacement 来代替匹配到的 regexp
${string/%regexp/replacement}	如果 string 的后缀匹配 regexp，那么就用 replacement 来代替匹配到的 regexp

例如，看看 morning 这个字符串的长度：

```
$ str=morning                    # 将字符串 morning 赋给 str
$ echo ${#str}
7
```

从 morning 这个字符串的第 0 个字符开始取子字符串，就是它本身：

```
$ echo ${str:0}
```

```
morning
```

分别从第 1 个字符和第 5 个字符开始取子字符串：

```
$ echo ${str:1}
orning
$ echo ${str:5}
ng
```

从第 2 个字符开始取长度为 3 的子字符串：

```
$ echo ${str:2:3}
rni
```

表 4−5 中的 regexp 是一个正则表达式，而不是普通的字符串，看下面的例子就明白了。

*代表零个或者多个字符，对于字符串 morning 而言，m*n 可以匹配 morn，也可以匹配 mornin。从字符串 morning 的开头, 删除最短匹配 m*n 的子串，也就是删除 morn，得到 ing：

```
$ echo ${str#m*n}
ing
```

从字符串 morning 的开头, 删除最长匹配 m*n 的子串，也就是删除 mornin，得到 g：

```
$ echo ${str##m*n}
g
```

从字符串 morning 的结尾, 删除最短匹配 n*g 的子串，也就是删除 ng，得到 morni：

```
$ echo ${str%n*g}
morni
```

从字符串 morning 的结尾, 删除最长匹配 n*g 的子串，也就是删除 ning，得到 mor：

```
$ echo ${str%%n*g}
mor
```

由上可知，井号（#）“控制着”字符串的开头（左边），百分号（%）“控制着”字符串的尾部（右边）。这不太容易记住，有时候会记反。这里提供一种记忆方法：#看上去像两个+错开放在一起，数的正号负号都是放在开头，如，+78；而写百分比时，%都是放在结尾，如，96%。或者看看键盘，#键在左边，%键在#的右边。

下面介绍#和%的应用，看一个“掐头去尾”的例子。已知一个文件的全路径，如，/usr/bin/zip，如果只想知道文件名 zip 本身，从头开始匹配，*/最长可以匹配/usr/bin/（其中，*匹配/usr/bin，/匹配 bin 后边的斜杠），“掐头”后，剩下 zip：

```
$ full_path=/usr/bin/zip
$ echo ${full_path##*/}
zip
```

如果只想知道文件的路径，从后面开始匹配，/*最短可以匹配/zip（其中，/匹配 zip 前面的斜杠，*匹配 zip），“去尾”后，剩下/usr/bin：

```
$ echo ${full_path%/*}
/usr/bin
```

上面的“掐头去尾”只是例子。实际工作中，几乎都使用 basename 和 dirname 这一对命令来得到文件名和文件的路径，顺便在这里介绍一下它们。命令 basename 的输入参数为一个带路径的文件时，输出为文件名本身，如：

```
$ basename /usr/bin/printf
printf                              # 路径去掉了，只剩文件名
$ basename ./my_shell
my_shell                            # 当前路径./去掉了
```

命令 dirname 的输入参数为一个带路径的文件时，输出为其路径，如：

```
$ dirname /usr/bin/printf
/usr/bin
$ full_path=/usr/bin/zip
$ basename $full_path
zip
$ dirname $full_path
/usr/bin
```

下面介绍表 4-5 中最后四行的内容。

给 str 重新赋值，mo 重复了三次：

```
$ str=momomorning
```

把第一个 mo 替换为 xyz：

```
$ echo ${str/mo/xyz}
xyzmomorning
```

把所有的 mo 替换为 xyz：

```
$ echo ${str//mo/xyz}
xyzxyzxyzrning
```

m*i 匹配 momomorni，把字符串前面的匹配 m*i 的部分替换为 xyz：

```
$ echo ${str/#m*i/xyz}
xyzng
```

i*g 匹配 ing，把尾部的匹配 i*g 的部分替换为 xyz：

```
$ echo ${str/%i*g/xyz}
momomornxyz
```

因为 mo 不在字符串 momomorning 的尾部，所以下面的操作不能做任何替换，得到的字符串与原字符串相同：

```
$ echo ${str/%mo/xyz}
momomorning
```

表 4-5 的前三行不仅适用于单个变量，也适用于数组。例如：

```
$ ARRAY=(one two three four five six seven eight)
$ echo ${#ARRAY[*]}                # 返回数组元素个数，即数组长度
8
$ echo ${ARRAY[*]}
one two three four five six seven eight
$ echo ${ARRAY[*]:3}               # 显示下标为 3 及其之后的所有元素
four five six seven eight
$ echo ${ARRAY[*]:3:2}             # 显示下标为 3 及其之后的 2 个元素
four five
```

表 4-5 的其他操作用在数组上的话，就是对每个数组元素进行同样的处理。例如：

```
$ ARRAY=(one two one three one four)
$ echo ${ARRAY[*]#one}
two three four
$ echo ${ARRAY[*]#t}
one wo one hree one four
$ echo ${ARRAY[*]#t*}
one wo one hree one four
$ echo ${ARRAY[*]##t*}
one one one four
```

4.17.2　用命令 expr 处理字符串

在字符串处理方面，expr 也“有所作为”，见表 4-6。注意表中的最后两行，小括号前面有反斜杠。由于小括号在命令行里有特殊的含义（小括号里面的命令是在子 shell 里面执行，见 3.11.8 节），所以使用的时候前面要加反斜杠屏蔽其特殊的功能。

表 4-6　expr 的字符串操作

命　　令	含　　义
expr length string	字符串 string 的长度
expr index string char	计算字符 char 在 string 中首次出现的位置（从 1 开始计），没找到的话返回 0
expr substr string pos length	从 string 的位置 pos 开始取长度为 length 的子串
expr match string regexp	string 开头的匹配 regexp 的长度
expr string : regexp	同上
expr match string \(regexp\)	从 string 的开头位置提取 regexp（注意，小括号前面有反斜杠）
expr string : \(regexp\)	同上

计算字符串的长度。例如，字符串 Good morning 的长度为 12：

```
$ expr length "Good morning"
12
```

查询某字符在字符串中首次出现的位置，例如：

```
$ expr index abcdefcc c
3                    # 3 表示 c 在字符串 abcdefcc 中首次出现在第 3 个字符
$ expr index abcdefcc k
0                    # 0 表示 k 未在字符串 abcdefcc 中出现
```

从某个字符串中取子串，例如，从“this is a book”的第 11 个字符开始，取出 4 个字符：

```
$ expr substr "this is a book" 11 4
book
```

表 4-6 中的 regexp 表示正则表达式。[a-z]表示任意一个小写字母，*表示匹配零次或者多次，那么[a-z]*可以匹配任意长度的小写字母串。字符串 abcde%123 的开头是长度为 5 的小写字母串：

```
$ expr match "abcde%123" "[a-z]*"
5
$ expr "abcde%123" : "[a-z]*"
5
```

“.”代表任意字符，.*可以与任意长度的字符串匹配。所以，也可以用如下方式计算字符串长度：

```
$ expr "Good morning" : '.*'
12
```

下面不是计算 abcde%123 的开头的小写字母串的长度，而是将小写字母串提取出来：

```
$ expr match "abcde%123" '\([a-z]*\)'
abcde
$ expr "abcde%123" : '\([a-z]*\)'
abcde
```

还可以利用 expr 提取字符串的某部分内容。例如，有个文件的名字为 salary.doc：

```
$ file=salary.doc
```

下面显示不带扩展名的文件名：

```
$ expr "$file" : '\(.*\).doc'
salary
```

因为命令中正则表达式有小括号中的.*和.doc 两部分，.doc 匹配了文件名 salary.doc 的扩展名.doc，所以.*就只能匹配 salary。模式匹配有一个“规矩”，就是“照顾全局”，同时也“兼顾局部”，让大家“都有份”。.*可以匹配任意长度的字符串，如果.*匹配了 salary.doc，那么.doc 就“没份”了，所以.*就匹配了 salary。.*处在\(与\)之间，被提取出来，所以，上面的命令的输出结果为 salary。

下面命令中正则表达式只有.*，.*匹配了 salary.doc：

```
$ expr "$file" : '\(.*\)'
salary.doc
```

下面命令中正则表达式有.*和.gif 两部分，尽管.*可以匹配任意长度的字符串，但是.gif 无法匹配 salary.doc 的任一部分，所以整个匹配是失败的，运行结果为空，并且 expr 命令的退出状态为 1（匹配失败）：

```
$ expr "$file" : '\(.*\).gif'
                                        # 空行
$ echo $?
1
```

要显示文件名 salary.doc 的后缀，可用如下命令：

```
$ expr $file : '.*\.\(.*\)'
doc
```

命令中正则表达式有.*、\.和小括号中的.*三部分。其中\.代表小数点本身，不再代表任意字符。从模式匹配的“规矩”可知，.*匹配了 salary，\.匹配了 salary.doc 中的小数点，小括号中的.*匹配了 doc。整体匹配是成功的，命令的结果为：显示出小括号中的匹配（或者说，提取小括号中的匹配），所以上面命令的结果为 doc。

第 5 章　条件流程控制

现实生活中有很多条件判断的例子。例如，在 ATM 上取钱，输入密码正确时，ATM 会提示：请输入取现金额（100 的整数倍）；输入密码错误时，ATM 会提示：密码错误，请再次输入。如果某种编程语言没有条件控制功能，几乎不会有人使用它。Bash 有针对整数、字符串和文件的条件判断，还有 if 结构和 case 结构用来进行条件流程控制。

5.1　条件判断与 test 命令

Bash 的内置命令 test 用来进行条件判断，命令格式为：

```
test 条件表达式
```

或者用中括号形式，格式为：

```
[ 条件表达式 ]
```

如果条件满足（或者说条件成立），test 命令的退出状态为 0；如果不满足，退出状态为 1。用中括号形式时，条件表达式与前后中括号之间要有空格。

5.1.1　整型数关系运算

两个整数之间有 6 种关系，见表 5-1。中括号形式的条件判断用得较多，所以本章的表格中，条件表达式都用中括号括起来了。下面表里的条件表达式，参与运算的必须是两个元素，这样的表达式叫做二元表达式，相应的运算符为二元运算符。

表 5-1　整型数关系运算

整 数 测 试	测 试 内 容
[int1 -lt int2]	int1 小于 int2
[int1 -le int2]	int1 小于或等于 int2
[int1 -eq int2]	int1 等于 int2
[int1 -ge int2]	int1 大于或等于 int2
[int1 -gt int2]	int1 大于 int2
[int1 -ne int2]	int1 不等于 int2

例如，10 小于 20，满足判断条件，下面命令的退出状态为 0：

```
$ test 10 -lt 20
$ echo $?
0
```

又如，15 不等于 25，15 -eq 25 是不成立的，所以下面命令的退出状态为 1：

```
$ i=15 k=25
$ [ $i -eq $k ]
$ echo $?
1
```

5.1.2　字符串关系运算

两个字符串之间的关系运算有等于、不等于、大于和小于，单个的字符串有两种判断：空与非空，见表 5−2。

表 5−2　字符串关系运算

字符串测试	测试内容
[str1 = str2]	str1 等于 str2
[str1 != str2]	str1 不等于 str2
[str1 < str2]	str1 小于 str2（按字典顺序, str1 在 str2 之前）
[str1 > str2]	str1 大于 str2（按字典顺序, str1 在 str2 之后）
[-z str]	str 为空串（zero），长度等于 0
[-n str]	str 非空（non-zero），长度大于 0
[str]	str 非空，长度大于 0

注意，表 5−2 中的运算符=，!=，>和<的两边必须有空格。在 Bash2 以上的版本，判断两个字符串是否相等用一个等号或者两个等号都可以。较老的 Bash 版本，必须用两个等号。现在，很难遇到安装了 Bash2 以下版本的 Linux 操作系统，所以不必考虑使用一个还是两个等号。但是可以相信，习惯使用 C 语言的人肯定喜欢使用两个等号。

例如，给 name 赋值为 Mike，则$name = Mike 是成立的，$name != Mike 是不成立的：

```
$ name=Mike
$ [ $name = Mike ]
$ echo $?
0
$ [ $name != Mike ]
$ echo $?
1
```

如果给 name 赋值为空格加 Mike，看看会怎样：

```
$ name=" Mike"
$ [ $name = Mike ]
$ echo $?
0
```

运行[$name = Mike]时，$name 被替换为空格加 Mike，相当于左中括号后面多了一个空格，命令变为[　Mike = Mike]。系统认为该命令在比较 Mike 与 Mike 是否相等，当然相

等，命令的退出状态为 0。这恐怕不是想得到的结果。

紧接着，运行["$name" = Mike]，$name 两边加上了双引号，系统认为该命令是比较空格加 Mike 与 Mike 是否相等，当然是不相等的，命令的退出状态为 1：

```
$ [ "$name" = Mike ]
$ echo $?
1
```

看来用不用引号，有时效果是不一样的，这一点需要注意。

表 5-2 中有字符串大小比较。按照字典顺序，排在后面的大，排在前面的小。比较的时候需要注意，由于>与<在命令行里表示重定向，所以使用大、小于号的时候，要在前面加上反斜杠屏蔽其特定含义。按照字典顺序，dog 排在 spring 的前面，“dog 小于 spring”是成立的：

```
$ [ dog \< spring ]
$ echo $?
0
```

按照字典顺序，apple 排在 banana 的前面，“apple 大于 banana”是不成立的：

```
$ str1="apple" str2="banana"
$ [ $str1 \> $str2 ]
$ echo $?
1
```

下面给 str 赋值为 abc，“字符串 str 为非空”是成立的：

```
$ str=abc
$ [ -n $str ]
$ echo $?
0
$ [ $str ]
$ echo $?
0
```

下面给 str 赋值为空（两个紧挨着的单引号，中间没有空格，表示空），“字符串 str 非空”是不成立的：

```
$ str=''
$ [ -n "$str" ]
$ echo $?
1
```

“字符串 str 为空”是成立的：

```
$ [ -z "$str" ]
$ echo $?
```

```
0
```

下面将 str 赋值为 3 个空格（两个单引号之间有 3 个空格），那么 str 是非空的。运行 test -z $str，来测试 str 是否为空。3 个空格被代入，命令变成 test -z 　　，与命令 test -z 效果一样，系统认为-z 的后面没有东西，str 被认为是空的，退出状态为 0，这显然不对：

```
$ str='   '
$ test -z $str
$ echo $?
0
```

再运行 test -z "$str"，来测试 str 是否为空。3 个空格被代入，命令变成 test -z "   "，系统认为-z 的后面有 3 个实实在在的空格，str 被认为是非空的，退出状态为 1，这才是对的：

```
$ test -z "$str"
$ echo $?
1
```

所以，在写脚本的时候要特别注意，对字符串的判断尽可能带上双引号。

在对变量进行比较的时候，根据“语境”，有时变量会变为字符串，有时会变为整数。下面给 str 赋值为 56，按照字典顺序，字符串“56”排在字符串“8”前面，“字符串 56 小于字符串 8”是成立的：

```
$ str="56"
$ [ $str \< 8 ]
$ echo $?
0
```

把 str 与整数相比较时，str 根据自己所处的环境，像变色龙一样，马上变成了整数 56，整数 56 大于整数 8，即 56 -gt 8 是成立的：

```
$ [ $str -gt 8 ]
$ echo $?
0
```

将一个纯字符串与整数相比，肯定是不合理的，比较时会遇到提示：

```
$ str=abcd
$ [ $str -gt 8 ]
bash: [: abcd: integer expression expected
```

5.1.3 文件属性条件判断

有时需要判断文件的属性。例如，有时进行某种操作之前要先确保某个文件存在，有时还要确保某个文件是所希望的某种类型。有关文件属性的条件运算，见表 5-3，表格里的条件表达式大多是一元的，最后三个是二元表达式。

表 5-3　文件属性运算

文件属性测试	测 试 内 容
[-a file]	file 存在
[-b file]	file 存在并且是块专用文件
[-c file]	file 存在并且是字符专用文件
[-d file]	file 存在并且是一个目录
[-e file]	file 存在，与-a 相同
[-f file]	file 存在并且是一个普通文件
[-g file]	file 存在并且设置了 SGID
[-h file]	file 存在并且是符号链接
[-k file]	file 存在并且设置了粘滞位
[-p file]	file 存在并且为已命名管道
[-r file]	file 存在并且当前用户具有读权限
[-s file]	file 存在并且非空（字节数大于 0）
[-t FD]	文件描述符 FD 打开并且指向一个终端
[-u file]	file 存在并且设置了 SUID
[-w file]	file 存在并且当前用户具有写权限
[-x file]	file 存在并且当前用户具有执行权限，如果 file 是目录，有进入该目录权限
[-L file]	file 存在并且是符号链接，与-h 相同
[-N file]	file 存在并且自最后一次读取后有更改
[-O file]	file 存在并且所有者的 UID 匹配当前用户的有效 UID（EUID），一般可理解为：当前用户是 file 的所有者
[-G file]	file 存在并且所属组的 GID 匹配当前用户的有效 GID，一般可理解为：当前用户首要组与 file 的所属组相同
[-S file]	file 存在并且是一个套接字
[file1 -nt file2]	根据修改时间 file1 比 file2 新（newer than），或者 file1 存在而 file2 不存在
[file1 -ot file2]	根据修改时间 file1 比 file2 老（older than），或者 file2 存在而 file1 不存在
[file1 -ef file2]	file1 是 file2 有相同的设备和节点号（互为硬链接）

例如，下面的文件 hello.mk，用户对该文件有读写权限，没有执行权限：

```
$ ls -l hello.mk
-rw-rw-r-- 1 user user 89 Nov 13 19:19 hello.mk
```

用户对该文件有读权限，命令 test -r hello.mk 的退出状态为 0：

```
$ test -r hello.mk
$ echo $?
0
```

用户对该文件有写权限，命令[-w hello.mk]的退出状态为 0：

```
$ [ -w hello.mk ]
$ echo $?
0
```

用户对该文件没有执行权限，命令[-x hello.mk]的退出状态为 1：

```
$ [ -x hello.mk ]
$ echo $?
1
```

接 2.9 节的内容，因为 p_hard_link.txt 是 b.txt 的硬链接，所以下面命令的退出状态为 0：

```
$ [ p_hard_link.txt -ef b.txt ]
$ echo $?
0
```

因为 p_hard_link.txt 与 b.txt 互为硬链接，所以下面命令的退出状态也为 0：

```
$ [ b.txt -ef p_hard_link.txt ]
$ echo $?
0
```

5.1.4　逻辑的与或非

有时需要判断两个条件是否同时成立。这时，运行：

```
test  条件表达式 1 -a  条件表达式 2
```

或者运行：

```
[  条件表达式 1 -a  条件表达式 2 ]
```

条件 1 与条件 2 都成立时，整条命令的退出状态为 0，否则为 1。这里的-a 是 and（与、并且）的意思。

有时，需要两个条件中的一个成立即可。这时，运行：

```
test  条件表达式 1 -o  条件表达式 2
```

或者运行：

```
[  条件表达式 1 -o  条件表达式 2 ]
```

条件 1 成立或者条件 2 成立（至少有一个成立），整条命令的退出状态为 0；条件 1 与条件 2 都不成立，整条命令的退出状态为 1。这里的-o 是 or（或者）的意思。

有时需要对某个条件的反面进行判断，格式为：

```
test !  条件表达式
```

或者为：

```
[ !  条件表达式  ]
```

条件不成立时，命令[！ 条件表达式]的退出状态为 0；条件成立时，退出状态为 1。与[条件表达式]的退出状态刚好相反。

本小节刚刚讲过的内容对应于表 5-4 的前三行。

表 5-4 与或非运算

与 或 非	测 试 内 容
[condition1 -a condition2]	condition1 与 condition2 都成立
[condition1 -o condition2]	condition1 与 condition2 至少有一个成立
[! condition]	condition 不成立（它的反面成立）
[[condition1 && condition2]]	condition1 与 condition2 都成立
[[condition1 || condition2]]	condition1 与 condition2 至少有一个成立
[[! condition]]	condition 不成立（它的反面成立）
[condition1] && [condition2]	condition1 与 condition2 都成立（这里用单、双中括号均可）
[condition1] || [condition2]	condition1 与 condition2 至少有一个成立（这里用单、双中括号均可）
! [condition]	condition 不成立（它的反面成立）（这里用单、双中括号均可）

下面看逻辑与、逻辑或、逻辑非的例子。

例如，上一小节提到过文件 hello.mk。如果要判断用户对该文件是否既有读权限又有写权限，可运行如下命令：

```
$ [ -r hello.mk -a -w hello.mk ]
$ echo $?
0
```

$?的值为 0，表明用户对文件 hello.mk 有读写权限。如果要判断用户对该文件是否既有读权限又有执行权限，运行如下命令：

```
$ [ -r hello.mk -a -x hello.mk ]
$ echo $?
1
```

$?的值为 1，表明“用户对文件 hello.mk 既有读权限又有执行权限”是不成立的。

如果要判断用户对文件 hello.mk 要么有读权限，要么有执行权限，或者都有，运行如下命令：

```
$ test -r hello.mk -o -x hello.mk
$ echo $?
0
```

$?的值为 0，表明“用户对文件 hello.mk 有读权限”与“用户对文件 hello.mk 有执行权限”，这二者至少有一个是成立的。

“10 大于 20”是不成立的：

```
$ x=10
$ [ $x -gt 20 ]
$ echo $?
1
```

“10 大于 20”的反面（逆命题）是成立的：

```
$ [ ! $x -gt 20 ]
$ echo $?
0
```

5.1.5　与或非的优先级

与、或、非可以任意组合，优先级由高到低分别是非、与、或，括号最优先，这里的括号指的是小括号。例如，前面的例子，加上小括号会更清楚一些，注意小括号与表达式之间有空格，小括号前面要加反斜杠（加反斜杠的原因在 4.17.2 节已经解释过）。

```
$ [ ! \( $x -gt 20 \) ]
$ echo $?
0
```

下面的例子，可以说明“与”比“或”的优先级高。

```
$ x=5 y=10 z=20                    #  三个赋值，用空格分隔
$ [ $x -eq 5 -o $y -gt 5 -a $z -lt 5 ]
$ echo $?
0
```

因为-a（与）的优先级比-o（或）高，$y -gt 5 -a $z -lt 5 先被判断，“y 大于 5”是成立的，“z 小于 5”不成立，所以$y -gt 5 -a $z -lt 5 不成立；接着$x -eq 5 被判断，“x 等于 5”成立，与成立的条件进行-o（或）运算，最终总是成立的，所以命令退出状态为 0。

用括号可以改变逻辑运算顺序，因为括号最优先。见下例，让“或”先得到运算机会。“x 等于 5”成立，所以括号里面的条件是成立的；“z 小于 5”不成立，与不成立的条件进行-a（与）运算，最终是不成立的，所以命令退出状态为 1：

```
$ [ \( $x -eq 5 -o $y -gt 5 \) -a $z -lt 5 ]
$ echo $?
1
```

5.1.6　双中括号格式

按照字典顺序，字符串 abc 排在 efg 的前面，并且，“10 小于或等于 20”是成立的，所以下面命令的退出状态为 0。为了逻辑更清楚，加上了小括号：

```
$ [ \( "abc" \< "efg" \) -a \( 10 -le 20 \) ]
$ echo $?
0
```

上面命令包含的反斜杠太多了，不易阅读。有什么改进的办法呢？见表 5-4 的中间三行，两层中括号是对一层中括号的扩展，使用双中括号，遇到大于号、小于号和小括号时，其前面不必再加反斜杠。使用双中括号时，与运算符是&&，或运算符是||，学过 C 语言的人

会感觉很熟悉。如果用双中括号，上面的例子变为：

```
$ [[ "abc" < "efg" && 10 -le 20 ]]
$ echo $?
0
```

再将两个表达式各自加上小括号，逻辑会更清晰，不需要使用反斜杠：

```
$ [[ ("abc" < "efg") && (10 -le 20) ]]
$ echo $?
0
```

再看一个例子。按照字典顺序，字符串 abc 排在 xyzw 的前面，“字符串 abc 大于字符串 xyzw”不成立，并且，“100 大于 200”也是不成立的，所以下面命令的退出状态为 1：

```
$ [[ ("abc" > "xyzw") || (100 -gt 200) ]]
$ echo $?
1
```

判断用户对文件 hello.mk 是否同时具有读权限和执行权限：

```
$ [[ ( -r hello.mk ) && ( -x hello.mk ) ]]
$ echo $?
1
```

双中括号比起单中括号，还有不一样的地方。在 5.1.2 节说过：对包含空格的字符串进行判断时要带上引号，否则判断结果可能不对。使用双中括号时，引号就不需要了，下面看两个例子。

接 5.1.2 节的例子，给 name 赋值为空格加 Mike，“变量 name 等于字符串 Mike”应该是不成立的。用双中括号判断时，$name 的两边不加引号，得到的判断结果也是对的：

```
$ name=" Mike"
$ [[ $name = Mike ]]
$ echo $?
1                    #“变量 name 等于 Mike”不成立，退出状态 1，判断结果正确
```

把 str 赋为三个空格，“变量 str 为空字符串”应该是不成立的。用双中括号判断时，$str 的两边不加引号，得到的判断结果也是对的：

```
$ str='   '
$ [[ -z $str ]]
$ echo $?
1                    # “变量 str 为空串”不成立，退出状态 1，判断结果正确
```

使用双中括号判断两个字符串相等或者不相等时，右边的字符串可以使用通配模式。例如，*代表零个或者多个字符，?代表一个字符。所以下面的前两个条件是成立的，退出状态为 0；最后一个不成立，退出状态为 1：

```
$ a=Wednesday
$ [[ $a = Wed* ]]
$ echo $?
0
$ [[ $a = Wed???day ]]
$ echo $?
0
$ [[ $a = Wed??day ]]
$ echo $?
1
```

5.1.7　在双小括号里面进行整数比较

表 4-1 提到了 Bash 支持的运算符。整数的关系运算还可以放在双小括号里面进行，这时的运算符与 C 语言的一致，&&表示与运算，||表示或运算，!表示非，==判断是否相等，!=判断是否不等。

```
$ (( 5>6 ))
$ echo $?
1       # 5>6 不成立，所以退出状态为 1
$ (( 5>3 && 7>=6 ))
$ echo $?
0       # 5>3 与 7>=6 都成立，所以退出状态为 0
$ (( 5>6 && 7>=6 ))
$ echo $?
1       # 5>6 不成立，虽然 7>=6 成立，退出状态还是为 1
$ (( 5>13 || 7>6 ))
$ echo $?
0       # 5>13 不成立，但是 7>6 成立，所以退出状态为 0
$ (( 5>13 || 7>16 ))
$ echo $?
1       # 5>13 不成立，7>16 也不成立，所以退出状态为 1
$ k=10
$ (( $k == 10 ))
$ echo $?
0       # k 等于 10 成立，所以退出状态为 0
$ (( $k == 15 ))
$ echo $?
1       # k 等于 15 不成立，所以退出状态为 1
$ (( $k != 15 ))
$ echo $?
0       # k 不等于 15 成立，所以退出状态为 0
$ (( ! ($k == 15) ))
$ echo $?
0       # k 等于 15 的反面是成立的，所以退出状态为 0
```

5.1.8　命令的与或非

Linux 的任何两条命令都可以使用&&进行与运算，格式为：

```
命令 1 &&  命令 2
```

当两条命令相与时，命令 1 先执行，命令 1 的退出状态为 0（成功）时，命令 2 执行；命令 1 的退出状态为非 0（失败）时，命令 2 不执行；只有当两条命令的退出状态都为 0 时，整个命令的退出状态为 0，否则为非 0。

下面将 date 命令与 pwd 命令进行与运算：

```
$ date && pwd
Sun Dec  2 08:51:09 EST 2012            #  这是 date 命令的执行结果
/home/user                              #  这是 pwd 命令的执行结果
$ echo $?
0              #  两条命令正常执行，所以整个命令的退出状态为 0
```

假设输入命令时，不小心把 date 输成了 dete，dete 执行失败，pwd 命令不执行：

```
$ dete && pwd
dete: command not found                 # dete 执行失败，pwd 命令没有执行
$ echo $?
127                                     #  整个命令的退出状态为非 0（127）
```

在写脚本时，当命令 2 的运行依赖于命令 1 的成功运行时（命令 1 成功运行之后，命令 2 才能运行），通常写为：

```
命令 1 &&  命令 2
```

例如，如果文件 a.txt 存在，则删除它，不存在则不删除，可以这样执行：

```
$ [ -e a.txt ] && rm -f a.txt
```

不仅两条 Linux 命令可以进行与运算，多条 Linux 命令也可以进行与运算，格式为：

```
命令 1 &&  命令 2 && … &&  命令 N
```

这些命令依次执行，当某条命令失败时，其后的命令不再执行。

举个例子，希望将文件 a.txt 复制为 b.txt，运行 cp a.txt b.txt 即可。但是，如果希望这件事成功率高一点，可以先保证 a.txt 存在，再保证 b.txt 不存在（或者存在并有写权限），可以用 ls 命令查看。例如：

```
$ ls -l a.txt b.txt
ls: cannot access b.txt: No such file or directory     # b.txt 不存在
-rw-rw-r-- 1 user user 0 Mar  6 22:18 a.txt            # a.txt 存在
```

运行 ls 命令，根据屏幕信息可以判断文件是否存在。而在运行的脚本中可以用下面的命令进行判断：

```
$ [ -e a.txt ] && [ ! -e b.txt ] && cp a.txt b.txt
```

该命令先判断 a.txt 是否存在，如果存在，继续判断 b.txt 是否不存在，如果不存在，运行复制命令。如果希望复制的成功率再高一点，还应判断 a.txt 是否可读，改进后的命令为：[-r a.txt] && [! -e b.txt] && cp a.txt b.txt。

Linux 的任何两条命令可以使用||进行或运算，格式为：

```
命令 1 || 命令 2
```

当两条命令相或时，命令 1 先执行，命令 1 的退出状态为 0（成功）时，命令 2 不再执行；命令 1 的退出状态为非 0（失败）时，命令 2 得到执行机会；只有当两条命令的退出状态都为非 0 时，整个命令的退出状态为非 0，否则为 0。

下面将 date 命令与 pwd 命令进行或运算：

```
$ date || pwd
Sun Dec  2 09:15:17 EST 2012     # date 命令正常执行，pwd 命令不再运行
$ echo $?
0                                # date 命令正常执行了，所以整个命令的退出状态为 0
```

假设输入命令时，不小心把 date 输成了 dete，dete 执行失败，pwd 命令得到执行机会：

```
$ dete || pwd
dete: command not found          # dete 执行失败，pwd 命令得到执行机会
/home/user                       # 这是 pwd 命令的执行结果
$ echo $?
0                                # pwd 命令的正常执行了，所以整个命令的退出状态为 0
```

在写脚本时，可能有这样的情景：如果命令 1 失败，希望命令 2 运行；如果命令 1 成功，命令 2 无需运行时。通常写为：

```
命令 1 || 命令 2
```

举个例子，读入一个数，显示它的绝对值。非负数的绝对值是它本身，负数的绝对值是它的相反数：

```
$ declare -i d                   # 声明整型数 d
$ d=-50                          # 赋值为一个负数-50
$ [ $d -ge 0 ] || d=-$d          # d=-50，d>=0 不成立，则执行 d=-d
$ echo $d
50                               # d 的值为 50
$ d=80                           # 赋值为一个正数 80
$ [ $d -ge 0 ] || d=-$d          # d=80，d>=0 成立，则不执行 d=-d
$ echo $d
80                               # d 的值为 80
```

命令 declare -i d 是必须的，如果没有这条命令，echo $d 的结果将是--50，而不是 50。因为，变量默认被当成字符串处理，并不是给它赋了个整数，它就完全具有了整数属性，命令 declare -i d 运行后，它才完全具有整数属性。

不仅两条 Linux 命令可以进行或运算，多条 Linux 命令也可以进行或运算，格式为：

```
命令 1 || 命令 2 || … || 命令 N
```

这些命令依次执行。当某条命令执行成功时，其后的命令不再执行；当某条命令及之前的命令都执行失败时，其后的命令才得到执行的机会。

还可以对 Linux 命令进行非运算。一般常对条件判断命令进行非运算，例如：

```
$ ! [ 200 -gt 100 ]
$ echo $?
1      # 200 大于 100，中括号内条件成立，它的反面不成立，所以，退出状态为 1
```

表 5-4 的最后三行实际上是 Linux 命令的与、或、非，条件判断命令也是命令，当然可以进行与、或、非的运算。例如，下面是判断命令[200 -gt 100]和判断命令[["xyz" > "abc"]]相与：

```
$ [ 200 -gt 100 ] && [[ "xyz" > "abc" ]]
$ echo $?
0    # 整数 200 大于 100，并且字符串 xyz 大于字符串 abc，所以退出状态为 0
```

下面是整数关系判断命令((5>3))与((7<=6))相或：

```
$ (( 5>3 )) || (( 7<=6 ))
$ echo $?
0      # 5>3 成立，所以退出状态为 0
```

5.1.9　判断变量是否定义

确认一个变量是否有定义，可以运行 set 命令。如果从 set 的输出中能找到该变量，就说明它有定义。set 命令的输出往往较长，从长长的输出中查找某个变量很不方便，借助 grep 命令，会方便一些。下面介绍一种简便的判断变量是否有定义的方法（Bash4.1 及以下的版本不支持该方法）。

判断一个变量是否定义的命令格式为：[-v 变量]，或者为：test -v <变量>，退出状态为 1 时，说明没有定义该变量，退出状态为 0 时，说明该变量有定义。对于普通变量和环境变量，此判断方法均有效。例如：

```
$ [ -v PS1 ]                # 系统提示符——环境变量 PS1 有定义
$ echo $?
0
$ a=98                      # 定义变量 a
$ [ -v a ]
echo $?
0
$ unset a                   # 取消定义
$ [ -v a ]
$ echo $?
1
```

5.2　条件测试结构 if

条件判断 if 的基本格式如下：

```
if 命令
then
    命令（命令组）
fi
```

该结构可以写在同一行（这时关键字 then 和 fi 前面的分号不能省略），格式为：

```
if 命令; then  命令（命令组）; fi
```

if 后面的命令退出状态为 0 时，then 与 fi 之间的命令被执行；if 后面的命令退出状态为非 0 时，then 与 fi 之间的命令不被执行。其流程如图 5-1 所示。

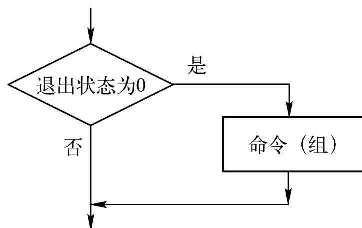


图 5-1　if 的流程图

在 if 结构中，常用中括号形式的条件判断命令，很少用 test 命令。例如，下面的脚本 score_1.sh，判断读入的分数，如果不低于 60 分则显示及格。

```
$ cat score_1.sh
#!/bin/bash
echo -n "please input your score: "        # 为用户显示提示
read score                                 # 键盘输入 score 值
if [ $score -ge 60 ]                       # 可用 test $score -ge 60，但中括号形式更常用
then
    echo Pass
fi
```

运行 score_1.sh，输入 86，则显示 Pass：

```
$ score_1.sh
please input your score: 86          # 在这里输入 86，不是另起一行输入，echo 选项-n 的作用
Pass
```

结合 5.1.8 节的内容，可以修改脚本 score_1.sh 为如下内容，效果是不变的：

```
$ cat score_1_same_result.sh
#!/bin/bash
```

```
echo -n "please input your score: "
read score
[ $score -ge 60 ] && echo Pass
```

有人会认为 if 的基本格式应该写为“if [条件判断]; then …”，而不是“if 命令; then …”。写为 if [条件判断]不能算错，也更容易理解。但是，if 后面可以是任何命令，不一定都是条件判断命令，只要命令的退出状态为 0（成功），then 与 fi 之间的命令就被执行。下面举例说明，当前目录下有文件 paper1.txt：

```
$ ls -l paper*.txt
-rw-rw-r-- 1 user user 178 Nov 15 11:08 paper1.txt
```

脚本 paper_copy.sh 的作用是，将文件 paper1.txt 复制为 paper2.txt，如果复制成功则打印 copy successfully：

```
$ cat paper_copy.sh
#!/bin/bash
if cp paper1.txt paper2.txt
then
    echo "copy successfully"
fi
```

运行脚本 paper_copy.sh，屏幕输出 copy successfully，说明复制成功了：

```
$ paper_copy.sh
copy successfully
```

查询一下，果然复制成功了：

```
$ ls -l paper*.txt
-rw-rw-r-- 1 user user 178 Nov 15 11:08 paper1.txt
-rw-rw-r-- 1 user user 178 Nov 15 11:10 paper2.txt
```

脚本 paper_copy.sh 中的 cp 命令也有可能失败，这时脚本不会打印 copy successfully，但是 cp 命令失败的信息会显示出来。例如，假设 paper1.txt 不存在，会显示如下信息：

```
$ paper_copy.sh
cp: cannot stat `paper1.txt': No such file or directory
```

如果用户希望：复制成功就看见 copy successfully，不成功就什么也不显示，则将标准错误重定向到“黑洞”即可，脚本改为：

```
$ cat paper_copy_2_null.sh
#!/bin/bash
if cp paper1.txt paper2.txt 2>/dev/null
then
    echo "copy successfully"
fi
```

5.3　if-else 结构

条件判断 if-else 结构提供了二路决策操作，它的基本格式如下：

```
if 命令
then
    命令（命令组）
else
    命令（命令组）
fi
```

或者写在同一行，格式如下：

```
if 命令; then  命令（命令组）; else  命令（命令组）; fi
```

if 后面的命令退出状态为 0 时，then 后面的命令被执行；if 后面的命令退出状态为非 0 时，else 后面的命令被执行。其流程如图 5-2 所示。

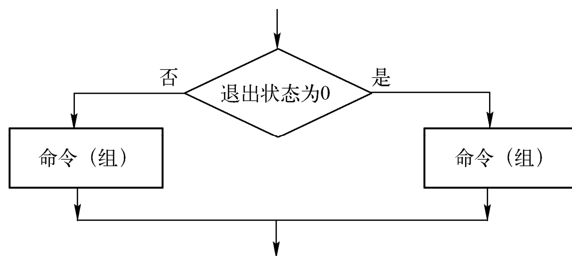


图 5-2　if-else 流程图

加强前面的脚本功能，得到脚本 score_2.sh，判断读入的分数，如果不低于 60 分则显示及格，否则显示失败。

```
$ cat score_2.sh
#!/bin/bash
echo -n "please input your score: "
read score
if [ $score -ge 60 ]
then
    echo Pass
else
    echo Fail
fi
```

运行脚本 score_2.sh 两次，分别输入 87 和 43，输出分别为 Pass 和 Fail：

```
$ score_2.sh
please input your score: 87
Pass
```

```
$ score_2.sh
please input your score: 43
Fail
```

结合 5.1.8 节的内容，可以修改脚本 score_2.sh 如下，效果是不变的。

```
$ cat score_2_same_result.sh
#!/bin/bash
echo -n "please input your score: "
read score
[ $score -ge 60 ] && echo Pass
[ $score -ge 60 ] || echo Fail
```

结合 5.1.8 节的内容，再考虑到“与”运算的优先级高于“或”运算，可以进一步修改脚本 score_2.sh 如下，效果是不变的。

```
$ cat score_2_same_effect.sh
#!/bin/bash
echo -n "please input your score: "
read score
[ $score -ge 60 ] && echo Pass || echo Fail
```

一般地，if-else 结构，if command1; then command2; else command3; fi 可以用 command1 && command2 || command3 代替。当 command2 与 command3 是单个命令，特别是比较短的单个命令时，这样替换是可以的；如果 command2 或 command3 是命令组，或者是很长的命令，最好不要这样替换，还是用 if-else 结构，可读性会好一些，清晰一些。

无论是 if 结构，if-else 结构，还是后面要讲的其他判断和循环结构，都是可以相互嵌套的。例如下面的脚本 score_3.sh，60 分以下显示 Fail，60 分及 60 分以上又分两种情况：不足 90 分的显示 Pass，大于或等于 90 分的显示优秀（Excellent）：

```
$ cat score_3.sh
#!/bin/bash
echo -n "please input your score: "
read score
if [ $score -lt 60 ]          # 外层 if-elese
then
    echo Fail
else
    if [ $score -lt 90 ]      # 嵌套的 if-else
    then
        echo Pass
    else
        echo Excellent
    fi
fi
```

运行脚本 score_3.sh 三次，分别输入 43、87、96，得到相应的输出：

```
$ score_3.sh
please input your score: 43
Fail
$ score_3.sh
please input your score: 87
Pass
$ score_3.sh
please input your score: 96
Excellent
```

再细分，如果将 60 分以下的显示 Fail，60（含）到 80 分的显示 Pass，80（含）到 90 分的显示 Very good，90（含）以上的显示 Excellent，继续使用嵌套的方法，那么判断脚本如下：

```
$ cat score_3_more_grade.sh
#!/bin/bash
echo -n "please input your score: "
read score
if [ $score -lt 60 ]; then
    echo Fail
else
    if [ $score -lt 80 ]; then
        echo Pass
    else
        if [ $score -lt 90 ]; then
            echo "Very good"
        else
            echo Excellent
        fi
    fi
fi
```

上面的脚本读起来不太容易。下面将这个脚本中成对的 if-else-fi 做上标记，逻辑层次就清晰了：

```
if [ $score -lt 60 ]; then
    echo Fail
else
    if [ $score -lt 80 ]; then
        echo Pass
    else
        if [ $score -lt 90 ]; then
            echo "Very good"
        else
            echo Excellent
        fi
    fi
fi
```

如果将判断再分得细一点，那么嵌套会越来越深，阅读和维护都不方便。此时使用接下来要讲的 if-elif 结构，逻辑会更清晰。

5.4　if-elif 结构

条件判断 if-elif 结构提供了多路决策操作，它的基本格式如下：

```
if 命令 1
then
    命令（命令组）
elif 命令 2
then
    命令（命令组）
[elif 可以有一个或者多个]
    ……
else
    命令（命令组）
fi
```

if 后面的命令 1 退出状态为 0（成功）时，then 后面的命令被执行；if 后面的命令 1 退出状态为非 0（失败）时，elif 后面的命令 2 被执行，并根据结果采取相应的动作，当命令 2 的退出状态为……，即开始新一轮的判断。elif 是 else if 的意思，但不能写为 else if。其流程如图 5-3 所示。

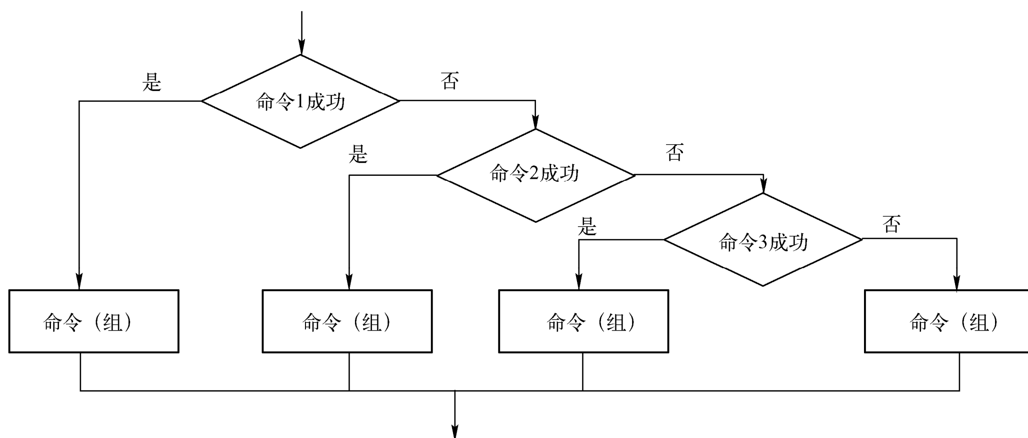


图 5-3　if-elif 流程图

举个例子就更清楚了。继续增强前面的脚本，判断读入的分数，不低于 90 分显示优秀，不低于 80 分显示良好，不低于 60 分显示及格，60 以下显示失败。

```
$ cat score_4.sh
#!/bin/bash
echo -n "please input your score: "
read score
```

```
if [ $score -ge 90 ]
then
    echo Excellent
elif [ $score -ge 80 ]
then
    echo "Very good"
elif [ $score -ge 60 ]
then
    echo Passing
else
    echo Failed
fi
```

运行该脚本四次，分别输入 95、86、78 和 56，输出分别是 Excellent、Very good、Passing 和 Failed：

```
$ score_4.sh
please input your score: 95
Excellent
$ score_4.sh
please input your score: 86
Very good
$ score_4.sh
please input your score: 78
Passing
$ score_4.sh
please input your score: 56
Failed
```

在 if-elif 结构中的 elif 可以有一个或者多个，else 最多只能有一个，else 部分不是必须的，可以没有。例如，将脚本 score_4.sh 中的 else 那部分的内容去掉，改为：

```
$ cat score_4_without_else.sh
#!/bin/bash
echo -n "please input your score: "
read score
if [ $score -ge 90 ]
then
    echo Excellent
elif [ $score -ge 80 ]
then
    echo "Very good"
elif [ $score -ge 60 ]
then
    echo Passing
fi
```

那么，输入 95、86、78 时，输出分别是 Excellent、Very good、Passing；输入 56 时，没有输出。可以认为脚本 score_4_without_else.sh 的意思是：60 分以下没有成绩。

判断某一年是不是闰年的方法是：能被 400 整除的是闰年，如 2000 年；能被 4 整除但不能被 100 整除的是闰年，如 2004 年；其余的都是平年，例如 2013 年是平年，1900 年是平年（因为它不能被 400 整除，能被 4、又能被 100 整除）。脚本 leap_year_1.sh 如下：

```
$ cat leap_year_1.sh
#!/bin/bash
read -p "Input year: " year
if [[ $[$year % 400] -eq 0 ]]; then
    echo This is a leap year.
elif [[ $[$year % 4] -eq 0 && $[$year % 100] -ne 0 ]]; then
    echo This is a leap year.
else
    echo This is not a leap year.
fi
```

测试一下：

```
$ leap_year_1.sh
Input year: 2008
This is a leap year.
$ leap_year_1.sh
Input year: 2013
This is not a leap year.
```

整数关系运算可以放在双小括号里面，修改一下，得到脚本 leap_year_2.sh：

```
$ cat leap_year_2.sh
#!/bin/bash
read -p "Input year: " year
if (( $year % 400 == 0 )) || (( ($year % 4 == 0) && ($year % 100 != 0) ))
then
     echo This is a leap year.
else
     echo This is not a leap year.
fi
```

测试一下：

```
$ leap_year_2.sh
Input year: 2012
This is a leap year.
$ leap_year_2.sh
Input year: 1700
This is not a leap year.
$ leap_year_2.sh
```

```
Input year: 2018
This is not a leap year.
```

5.5　分情况选择处理——case 命令

从前面的内容可知，如果要对两种情况进行处理，可以使用 if-else 结构。如果要对多种情况进行处理，可以使用 if-elif-else 结构（或者用嵌套的 if-else）。如果情况比较多时，if-elif-else 将变得很长（或者 if-else 的嵌套层数变得很多），也不容易理解。对多种情况进行处理，用具有选择功能的 case 命令更加合适。它的语法格式如下：

```
case  变量 in
    模式 1 [ |  模式 1.2 [ |  模式 1.3 ]…)
        命令或命令组;;
    模式 2 [ |  模式 2.2 [ |  模式 2.3 ]…)
        命令或命令组;;
    模式 3 [ |  模式 3.2 [ |  模式 3.3 ]…)
        命令或命令组;;
    …
esac
```

注意，case 命令结束的关键字是 esac，也就是 case 的倒序，命令或命令组的后面是两个分号，两个分号不能省略。下面的例子，输入 1 到 7 的任意一个数字，打印星期一到星期日当中相应的日子。

```
$ cat case_select_1.sh
#!/bin/bash
echo -n please input a number from 1 to 7:
read day
case $day in
    1)  echo Monday
        ;;
    2)  echo Tuesday
        ;;
    3)  echo Wednesday
        ;;
    4)  echo Thursday
        ;;
    5)  echo Friday
        ;;
    6)  echo Saturday
        ;;
    7)  echo Sunday
        ;;
    *)  echo input error
        ;;
```

```
esac
```

运行结果如下：

```
$ case_select_1.sh
please input a number from 1 to 7:1        #  键盘输入 1 显示周一
Monday
$ case_select_1.sh
please input a number from 1 to 7:7        #  键盘输入 7 显示周日
Sunday
$ case_select_1.sh
please input a number from 1 to 7:9        #  输入 9 得到提示 input error
input error
$ case_select_1.sh
please input a number from 1 to 7:abc   #  输入非数字得到提示 input error
input error
```

当用户输入 1 到 7 以外的数字或者其他字符，将匹配*，所以显示 input error。该脚本对应的流程如图 5-4 所示。

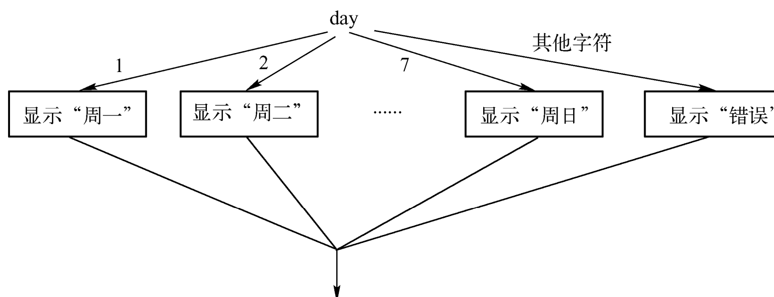


图 5-4　脚本 case_select_1.sh 的流程图

下面的脚本判断输入的分数值，100 则显示满分，90～99 则显示优秀，80～89 则显示良好，60～79 显示及格，其他显示不及格。

```
$ cat case_select_2.sh
#!/bin/bash
echo -n please input a number from 0 to 100:
read score
case $score in
    100)    echo "Full Mark"
            ;;
    9[0-9]) echo "Excellent"
            ;;
    8[0-9]) echo "Very Good"
            ;;
    [6-7][0-9])    echo "Passing"
            ;;
```

```
    *)   echo "Failed, or input error"
              ;;
esac
```

上例中，9[0-9]与 90-99 匹配，[6-7][0-9]与 60-79 匹配。学习正则表达式后，会更加清楚这一点。

运行结果如下：

```
$ case_select_2.sh
please input a number from 0 to 100:100        # 键盘输入 100
Full Mark
$ case_select_2.sh
please input a number from 0 to 100:72         # 键盘输入 72
Passing
$ case_select_2.sh
please input a number from 0 to 100:55         # 键盘输入 55
Failed, or input error
$ case_select_2.sh
please input a number from 0 to 100:abcd       # 键盘输入非数字
Failed, or input error
```

case 命令中的模式，可以只有一个模式（就像前面的例子），也可以是多个模式，各个模式之间是或的关系，用竖线分割。下面的脚本，提示用户键盘输入一个小写字母，如果输入的是 a，e，i，o 和 u 中的某个字母时（脚本里写为：a|e|i|o|u），显示“它是元音字母”，如果输入其他小写字母时，显示“它不是元音字母”，如果输入的不是小写字母时，显示“输入错误”：

```
$ cat case_vowel.sh
#!/bin/bash
echo -n please input a letter from a to z:
read letter
case $letter in
    a|e|i|o|u)
        echo "It is a vowel."
        ;;
    [b-d]|[f-h]|[j-n]|[p-t]|[v-z])
        echo "It is not a vowel."
        ;;
    *)   echo "input error."
        ;;
esac
```

下面运行脚本：

```
$ case_vowel.sh
please input a letter from a to z:u                # 键盘输入 u
```

```
It is a vowel.                          # 显示“它是元音字母”
$ case_vowel.sh
please input a letter from a to z:x     # 键盘输入 x
It is not a vowel                       # 显示“它不是元音字母”
$ case_vowel.sh
please input a letter from a to z:8     # 键盘输入非小写字母
input error                             # 显示“输入错误”
```

5.6 命令 exit 与 if 及 case 命令的配合

内置命令 exit 用来退出 shell。一个脚本里面有 exit 命令时，脚本运行到 exit 就退出了，exit 之后的其他命令不再执行。exit 命令的退出状态默认为它的前一条命令的退出状态；exit 命令也可以带一个整数参数，这时它的退出状态就是这个整数。

可以根据不同的情况，让一个脚本退出状态有相应的值，每个值有自己的含义。例如，某专业的社会考试的报考条件为：博士可以直接报考；硕士，工作两年以上的可以报考，但毕业证上的专业与本考试的专业不符者，需要加试一门课；本科，工作五年以上的可以报考，但毕业证的专业与本考试的专业不符者，需要加试一门课；其他条件的人无报考资格。脚本 if_exit.sh 对考生资格进行审核，可以直接报考的用 1 表示，可以报考但需要加试的用 2 表示，不能报考的用 3 表示：

```
$ cat if_exit.sh
#!/bin/bash
if [[ "$degree" == Doctor ]]; then
    exit 1
elif [[ ("$degree" == Master) && ($work_life -ge 2) ]]; then
    if [ "$major" == statistics ]; then
      exit 1
    else
      exit 2
    fi
elif [[ ("$degree" == Bachelor) && ($work_life -ge 5) ]]; then
    if [ "$major" == statistics ]; then
      exit 1
    else
      exit 2
    fi
else
    exit 3
fi
```

脚本 enroll.sh 的参数有三个，调用格式为：“enroll.sh 学历 专业 工龄”。该脚本首先把三个参数存入相应的变量并导出，以便它的子 shell（被它调用的脚本）可以使用。然后，调用了脚本 if_exit.sh（如果这两个脚本在同一个目录，if_exit.sh 可以不带路径）。最后，根据

if_exit.sh 的退出状态，做出相应的处理。

```
$ cat enroll.sh
#!/bin/bash
export degree=$1          # 学历
export major=$2           # 专业
export work_life=$3       # 工龄

# 调用脚本 if_exit.sh，如果两个脚本不在同一个目录，if_exit.sh 需要带路径
if_exit.sh $degree $major $work_life

case $? in                # $?为上一条命令——脚本 if_exit.sh 的退出状态值
  1) echo OK
     ;;
  2) echo OK, but you must select statistics exam.
     ;;
  3) echo Sorry, you cannot enroll.
     ;;
  *) echo input error
     ;;
esac
```

运行脚本 enroll.sh 试试，结果和预期的相符：

```
$ enroll.sh Doctor math 3
OK                                        # 博士可报考
$ enroll.sh Master statistics 2
OK                                        # 工作 2 年专业相符的硕士可报考
$ enroll.sh Bachelor computer 6
OK, but you must select statistics exam.  # 工作 6 年专业不相符的本科加试后可报考
$ enroll.sh Master physics 1
Sorry, you cannot enroll                  # 工作 1 年的硕士不可报考
```

5.7　用 here 文档与 case 命令生成菜单

在 shell 脚本程序中向一条命令传递输入的一种特殊方法是使用 here 文档。它允许一条命令在获得输入数据时，就好像是在读取一个文件或用键盘输入一样，而实际上是从脚本程序中得到输入数据。here 文档格式：

```
命令 << SpecialString
......
SpecialString
```

here 文档以两个连续的小于号<<开始，紧跟着一个特殊的字符序列，该序列将在文档的结尾处再次出现。这个特殊字符序列的作用就像一个标记，它告诉系统 here 文档结束的位

置。因为这个标记序列不能出现在传递给命令的文档内容中，所以应该尽量使它既容易记忆又相当不寻常。

使用 here 文档，最简单也是最常见的例子就是给 cat 命令提供输入数据。见脚本 here.sh：

```
$ cat here.sh
#!/bin/bash
cat << SUNNY!
Hello Mike,
This is a here documnet.
Best regards,
Jack
SUNNY!
```

脚本 here.sh 使用特殊字符串 SUNNY!作为 here 文档的起止标志，运行脚本，输出如下：

```
$ here.sh
Hello Mike,
This is a here documnet.
Best regards,
Jack
```

如果要抑制 Tab 功能（去掉每行前面的 Tab 字符），将<<后面加上“-”即可，这时，here 文档格式：

```
命令  <<- SpecialString
......
SpecialString
```

下一章要讲的 select 命令用来生成菜单。用 here 文档与 case 命令，也可以生成菜单。例如，脚本 here_case.sh 用来显示几种 build 的环境，供用户选择：

```
$ cat here_case.sh
#!/bin/bash
cat << !BUILD_OS!
    1)Solaris
    2)Linux
    3)Cygwin
    4)Win32
!BUILD_OS!

read selection        # 需要选择 1-4，键盘输入 1 到 4 之间的某个数
case "$selection" in
    1) export BUILD_OS=Solaris
        ;;
    2) export BUILD_OS=Linux
```

```
        ;;
    3) export BUILD_OS=Cygwin
        ;;
    4) export BUILD_OS=Win32
        ;;
esac
echo "Now, Build OS is $BUILD_OS."
```

执行脚本 here_case.sh，看到由 cat 命令显示的 here 文档之后，键盘输入 1，则显示选择了 Solaris：

```
$ here_case.sh
    1)Solaris                    # 由 cat 命令输出菜单选项 1-4 供用户选择
    2)Linux
    3)Cygwin
    4)Win32
1                                # 键盘输入 1，export BUILD_OS=Solaris 被执行
Now, Build OS is Solaris.        # 脚本最后一句（echo 命令）的执行结果
```

执行脚本 here_case.sh，键盘输入 3，则显示选择了 Cygwin：

```
$ here_case.sh
    1)Solaris
    2)Linux
    3)Cygwin
    4)Win32
3                                # 键盘输入 3，export BUILD_OS=Cygwin 被执行
Now, Build OS is Cygwin.         # 脚本最后一句（echo 命令）的执行结果
```

5.8　null 命令

Bash 提供了内置的空命令，也叫 null 命令，它是个冒号，运行时，它什么也不做，退出状态总是成功。null 命令什么也不做，但在有些情况下，使用它是必要的。

脚本 null_1.sh 判断当前目录下是否有文件 book.txt，如果没有，显示“文件不存在”，如果有，什么也不显示，关键字 then 后面使用了一个冒号，即 null 命令：

```
$ cat null_1.sh
#!/bin/bash
if [ -f book.txt ]
then
    :                            # null 命令，即冒号
else
    echo "This file doesn't exist."
fi
```

运行脚本 null_1.sh，假设当前目录下没有文件 book.txt，将显示“文件不存在”：

```
$ null_1.sh
This file doesn't exist.
```

脚本 null_1.sh 中的 null 命令是必要的。如果将 then 后面的 null 命令变为空行：

```
$ cat null_1_blank_line.sh
#!/bin/bash
if [ -f book.txt ]
then
                    # 空行
else
    echo "This file doesn't exist."
fi
```

再运行脚本，会遇到错误：

```
$ null_1_blank_line.sh
./null_1_blank_line.sh: line 5: syntax error near unexpected token `else'
```

这是因为，如果脚本中 then 的后面是空行，关键字 then 和关键字 else 之间将没有任何命令，两个关键字“直接接触”了，所以出现语法错误。

在 4.15 节讲述 expr 命令时，讲过如何判断键盘输入的是整数，不是小数，也不是字母或字符串。脚本 null_2.sh 让键盘输入值存入变量 int，再使用命令 expr 让 int 与整数 10 进行加法运算，如果命令 expr 的退出状态是非 0，说明键盘输入的不是整数。

```
$ cat null_2.sh
#!/bin/bash
echo "Please input an integer"
read int
if expr "$int" + 10 >& /dev/null    # 将 expr 命令的输出重定向到“黑洞”
then
    :
else
    echo "This is not an integer."
    exit 1
fi
```

下面运行 null_2.sh 三次，分别输入整数、小数、字母：

```
$ null_2.sh
Please input an integer
6                          # 输入整数 6
$ null_2.sh
Please input an integer
7.8                        # 输入小数 7.8
```

```
This is not an integer.              # 显示这不是整数
$ null_2.sh
Please input an integer
f                                    # 输入字母
This is not an integer.              # 显示这不是整数
```

将脚本 null_2.sh 中的 if 后面取反，得到如下的 null_3.sh，这两个脚本的功能完全一致：

```
$ cat null_3.sh
#!/bin/bash
echo "Please input an integer"
read int
if ! expr "$int" + 10 >& /dev/null       # 将 expr 命令的输出重定向到“黑洞”
then
    echo "This is not an integer."
    exit 1
fi
```

null_2.sh 使用了 null 命令，比 null_3.sh“啰唆”，既然 null_3.sh 与 null_2.sh 作用相同，为什么不丢弃 null_2.sh 呢？实际上，写一个较长的脚本，通常不是马上可以完成的。有的地方暂时没有想好如何处理，或者相应的条件还未成熟，暂时没办法处理，可又希望脚本能尽快运行起来，这时就可以将 null 命令放置在暂时没有想好的地方。

null 命令还有一个特点，不但自身的退出状态总是 0，还可以带任何参数，并且退出状态总是 0。下面的 null 命令后面带了 5 个随机输入的、凌乱的参数：

```
$ : dog pig pencil + "xyz"
$ echo $?
0                                # null 命令退出状态为 0
```

如果将这 5 个凌乱的参数直接作为命令执行，肯定是失败的：

```
$ dog pig pencil + "xyz"
dog: command not found
```

null 命令的这一特点是可以利用的。先复习 4.16 节的内容，根据该节的知识，在表达式 ${var:=180}中，var 无值时，var 将被赋值为 180，有值的话将保持原来的值。

```
$ var=                               # 先给 var 赋空值，则 var 无值
$ : ${var:=180}                      # ${var:=180}作为 null 命令的参数
$ echo $var
180                                  # var 被赋值为 180
$ : ${var:=360}                      # ${var:=360}作为 null 命令的参数
$ echo $var
180                                  # 因为这时 var 有值，所以 var 保持原来的值 180
```

如果不将${var:=180}作为 null 命令的参数，直接运行${var:=180}，会出错，因为${var:=180}是一个表达式，不是一个完整的命令。该表达式的值是 180，所以，Linux 系统会认为用户在执行命令 180，这个命令是不存在的，运行出错：

```
$ ${var:=180}
180: command not found
```

将${var:=180}作为空命令的参数，即，在前面加一个冒号和空格，就不会出错了。

第 6 章　循　　环

生活中循环的例子很多，如车轮的转动、四季的变化、节目的反复播放等。Bash 支持循环功能，并可以控制循环，包括循环的次数、循环的跳出和结束、循环时变量如何变化（如递增或递减）等。

6.1　for 循环

for 循环的基本语法格式为：

```
for 变量 [ in 列表 ]
do
    命令（命令组）
done
```

或者为（在一行之内写 for 循环时注意，关键字 do 和 done 前面的分号不能省略）：

```
for 变量 [ in 列表 ] ; do 命令（命令组）; done
```

for 循环的流程如图 6-1 所示。

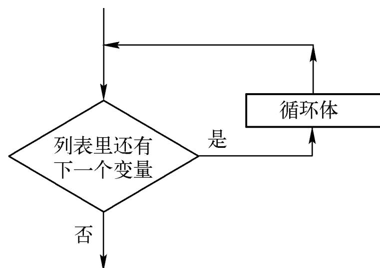


图 6-1　for 循环流程图

下面的例子是对 4 个人依次问候早安，循环 4 次。变量 name 的值在 4 次循环当中依次是 Tom、Jack、Harry 和 Merry。

```
$ cat for_loop.sh
#!/bin/bash
for name in Tom Jack Harry Merry
do
    echo "Good morning, $name."
done
```

运行该脚本，得到输出如下：

```
$ for_loop.sh
Good morning, Tom.
Good morning, Jack.
Good morning, Harry.
Good morning, Merry.
```

for 循环的变量列表还可以来自文件：

```
$ cat name_list.txt
Tom
Jack
Harry
Merry
```

脚本 for_cat.sh 中，使用了命令替换功能，将 cat name_list.txt 的输出作为 for 循环的变量列表：

```
$ cat for_cat.sh
#!/bin/bash
for name in $(cat name_list.txt)
do
    echo "Good morning, $name."
done
```

执行脚本 for_cat.sh，效果与脚本 for_loop.sh 相同：

```
$ for_cat.sh
Good morning, Tom.
Good morning, Jack.
Good morning, Harry.
Good morning, Merry.
```

for 循环也可以在命令行直接输入执行，不是必须放在一个脚本里面。下例打印出 1 到 5：

```
$ for i in 1 2 3 4 5
> do
> echo $i
> done
1
2
3
4
5
```

>为系统给出的续行提示符。若 for 循环可以在一行之内输入，就“不用麻烦”系统给出续行提示符了，如：

```
$ for i in 1 2 3 4 5; do echo $i; done
1
2
3
4
5
```

当 for 命令很长时，虽然仍旧可以在一行之内输入，但是可读性不好，将 for 命令分为多行，可读性则会好很多。

在表 4-3 中，内置变量 IFS，即字段分隔符（Internal Field Seperator），默认为空白符。上面关于 for 循环的例子中，变量列表中的元素间的分隔符是空格，实际上分隔符可以通过 IFS 来定义。下例中定义分隔符为冒号：

```
$ cat for_IFS.sh
#!/bin/bash
MY_NUM="1:2:3:4:5"
IFS=:
for i in $MY_NUM
do
    echo $i
done
```

运行结果如下：

```
$ for_IFS.sh
1
2
3
4
5
```

Bash 的内置变量 PATH 存放着搜索路径，路径之间是以冒号间隔的。如下脚本先打印搜索路径，再按照先后顺序将每个路径显示出来：

```
$ cat for_path.sh
#!/bin/bash
echo PATH=$PATH
IFS=:
for path in $PATH
do
    echo $path
done
```

运行 for_path.sh，得到了预期的结果：

```
$ for_path.sh
PATH=/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:.
```

```
/usr/lib/lightdm/lightdm
/usr/local/sbin
/usr/local/bin
/usr/sbin
/usr/bin
/sbin
/bin
/usr/games
.
```

如果循环 变量的取值是连续的，如取 1 到 6，则可以用{1..6}表示。例如，打印 1 到 6：

```
$ for i in {1..6}; do echo $i; done
1
2
3
4
5
6
```

打印 a 到 e：

```
$ for i in {a..e}; do echo $i; done
a
b
c
d
e
```

Bash 自己可以判断出是升序还是降序。如：

```
$ for i in {20..15}; do echo $i; done
20
19
18
17
16
15
```

顺便说一下，实际上，显示连续的数字或者字母，用 echo 命令就行。如：

```
$ echo {1..10}
1 2 3 4 5 6 7 8 9 10
$ echo {z..m}
z y x w v u t s r q p o n m
```

在讲述内置特殊变量的时候，提到过$@和$*，见表 4-2。不带双引号时，它们是一样的，带双引号时，它们是不一样的。例如，脚本 for_at_in_quotes.sh 使用了"$@"，那么脚本

for_at_in_quotes.sh 有多个参数时，参数被传入"$@"，每个参数是独立的。

```
$ cat for_at_in_quotes.sh
#!/bin/bash
for name in "$@"
do
    echo "Hello $name"
done
```

脚本 for_at_in_quotes.sh 带上 4 个参数，即 4 个人名，循环 4 次分别对他们说 Hello：

```
$ for_at_in_quotes.sh Tom Jack Harry Merry
Hello Tom
Hello Jack
Hello Harry
Hello Merry
```

脚本 for_star_in_quotes.sh 使用了"$*"，那么脚本 for_star_in_quotes.sh 有多个参数时，参数被传入"$*"，所有的参数被作为一个整体。

```
$ cat for_star_in_quotes.sh
#!/bin/bash
for name in "$*"
do
    echo "Hello $name"
done
```

脚本 for_star_in_quotes.sh 带上 4 个名字参数，只循环一次，对他们一起说 Hello：

```
$ for_star_in_quotes.sh Tom Jack Harry Merry
Hello Tom Jack Harry Merry
```

也就是说，带上同样的 4 个参数，传入"$@"时，脚本 for_at_in_quotes.sh 认为有 4 个循环变量值；传入"$*"时，4 个参数被当做一个整体，脚本 for_star_in_quotes.sh 认为只有一个循环变量值。

$@和$*不带双引号时，它们是一样的。for_at_no_quotes.sh 使用不带双引号的$@：

```
$ cat for_at_no_quotes.sh
#!/bin/bash
for name in $@
do
    echo "Hello $name"
done
```

传递 4 个人名参数给 for_at_no_quotes.sh：

```
$ for_at_no_quotes.sh Tom Jack Harry Merry
Hello Tom
```

```
Hello Jack
Hello Harry
Hello Merry
```

for_star_no_quotes.sh 使用不带双引号的$*：

```
$ cat for_star_no_quotes.sh
#!/bin/bash
for name in $*
do
    echo "Hello $name"
done
```

传递 4 个人名参数给 for_star_no_quotes.sh，可见，效果与 for_at_no_quotes.sh 一致：

```
$ for_star_no_quotes.sh Tom Jack Harry Merry
Hello Tom
Hello Jack
Hello Harry
Hello Merry
```

现在，可以较深刻地理解$@和$*的区别了。${数组名[@]}与${数组名[*]}的区别与$@和$*的区别，是一样的。

文件 name_list.txt 存放 4 个人名：

```
$ cat name_list.txt
Tom
Jack
Harry
Merry
```

创建数组 list，记录 4 个人名：

```
$ mapfile -t list < name_list.txt
```

查看一下数组 list：

```
$ declare -p list
declare -a list='([0]="Tom" [1]="Jack" [2]="Harry" [3]="Merry")'
```

然后对数组循环，对数组的每个元素值说 hello：

```
$ for name in "${list[@]}"; do echo Hello $name; done
Hello Tom
Hello Jack
Hello Harry
Hello Merry
```

如果使用"${数组名[*]}"，只循环一次，对所有的元素值一起说 hello：

```
$ for name in "${list[*]}"; do echo Hello $name; done
Hello Tom Jack Harry Merry
```

for 的基本格式当中的“in 列表”可以省略。省略以后，“for 变量”等同于“for 变量 in "$@"”。见脚本 for_no_list.sh：

```
$ cat for_no_list.sh
#!/bin/bash
for name
do
    echo "Hello $name"
done
```

脚本 for_no_list.sh 中的 for name 等同于 for name in "$@"，即脚本 for_no_list.sh 等同于脚本 for_at_in_quotes.sh：

```
$ for_no_list.sh Tom Sam David Emma
Hello Tom
Hello Sam
Hello David
Hello Emma
```

也就是说，当把脚本参数传给脚本内的 for 循环时，for 循环中的“in 列表”可以省略。但是，不省略时脚本的可读性会好一些。

6.2　算术 for 循环

算术 for 循环的基本格式如下：

```
for (( 表达式 1; 表达式 2; 表达式 3 )); do 命令（命令组）; done
```

这种 for 循环与 Java 及 C 语言的 for 循环思路是一致的。基本格式写成下面这样，更容易理解：

```
for (( 变量初始化; 循环的条件; 变量值更新/递增/递减 ))
do
    命令（命令组）
done
```

算术 for 循环的流程如图 6-2 所示。

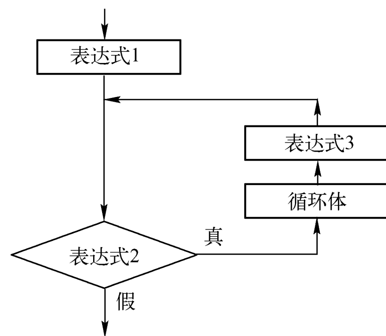


图 6-2　算术 for 循环的流程图

下面的脚本，变量 i 的初始值是 1，因为 1<=10 成立，所以循环可以进行，打印出 1，然后 i++，i 自增，i=2；因为 2<=10 成立，所以循环可以继续进行，打印出 2，然后 i++，i 自增，i=3；……；…，i=10，因为

10<=10 成立，所以循环可以继续进行，打印出 10，然后 i++，i 自增，i=11，11<=10 不成立，循环结束。

```
$ cat arithmetic_for_1.sh
#!/bin/bash
for((i=1;i<=10;i++))
do
    echo $i
done
```

运行的结果就是打印出 1 到 10：

```
$ arithmetic_for_1.sh
1
2
3
4
5
6
7
8
9
10
```

算术 for 循环的小括号之间的 3 个表达式都不是必须的，可以将变量初始化放在循环体的前面，将变量值的更新（如递增和递减）放在循环体之内，但是小括号之间的两个分号不能省略。例如，下面的脚本，效果与前面的完全一致：

```
$ cat arithmetic_for_2.sh
#!/bin/bash
i=1
for(( ; i<=10 ; ))
do
    echo $i
    ((i++))
done
```

运行结果与 arithmetic_for_1.sh 的完全一致：

```
$ arithmetic_for_2.sh
1
2
3
4
5
6
7
```

```
8
9
10
```

前面提到，算术 for 循环的小括号之间的 3 个表达式都不是必须的，循环条件表达式也可以不写，这时 for 循环成为了无条件循环，有可能成为无穷循环，俗称死循环。例如，下面的脚本是无穷循环：

```
$ cat arithmetic_for_3.sh
#!/bin/bash
for(( ; ; ))
do
    echo "Good morning"
done
```

运行 arithmetic_for_3.sh 将打印出无数个 Good morning，按〈Ctrl+C〉键可以终止它：

```
$ arithmetic_for_3.sh
Good morning
……
Good morning
Good morning^C
```

for((; ;))形式的循环不一定都是无穷循环，学到后面的 break 命令就知道了。

6.3　while 循环

while 循环也叫当型循环，格式如下，do 与 done 之间为循环体：

```
while 命令
do
    命令（命令组）
done
```

或者为：

```
while 命令; do 命令（命令组）; done
```

while 先判断它后面的命令的退出状态，退出状态为 0 时（命令成功时，当 while 循环的条件成立时），关键字 do 之后的命令开始执行，执行到关键字 done 后，关键字 while 后面的命令再次执行，如果退出状态还是 0（循环条件还是成立），循环将继续进行，即循环体内的命令继续执行……当 while 循环条件不再成立时，循环结束。

while 循环的流程如图 6-3 所示。

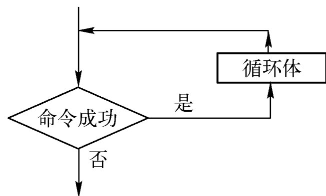


图 6-3　while 循环流程图

```
$ cat while_loop_1.sh
#!/bin/bash
i=1
while [ $i -le 3 ]
do
    echo "i="$i
    let i=$i+1
done
```

上面的例子，先将 i 的初值赋为 1，接着判断 i 是否小于或等于 3，因为 1 小于或等于 3，所以显示 i=1，再将 i 的值加 1；i 的值变为 2，因为 2 还是小于或等于 3，所以显示 i=2，再将 i 的值加 1；i 的值变为 3，因为 3 仍然小于或等于 3，所以显示 i=3，再将 i 的值加 1；i 的值变为 4，i 不再小于或等于 3，条件不再成立，循环结束，循环体内的命令不再执行。脚本的运行结果如下：

```
$ while_loop_1.sh
i=1
i=2
i=3
```

while 循环也可以是无穷循环。

```
$ cat while_infinite_loop.sh
#!/bin/bash
while [ 5 -le 10 ]
do
    echo "Nice to meet you"
done
```

上面的例子中，5 本来就小于 10，就是说循环条件永远都是成立的。运行该脚本会一直显示 Nice to meet you，可以按下〈Ctrl+C〉键强行终止它。

```
$ while_infinite_loop.sh
Nice to meet you.
Nice to meet you.
……
Nice to meet you.
Nice to meet ^C                # 用户按下〈Ctrl+C〉键
```

内置命令 true 的退出状态总是 0（成功），使用 while true 可以构成无穷循环。例如：

```
$ cat while_true_infinite_loop.sh
#!/bin/bash
while true
do
    echo "Nice to meet you."
done
```

运行 while_true_infinite_loop.sh，得到的结果与 while_infinite_loop.sh 的相同：

```
$ while_true_infinite_loop.sh
Nice to meet you.
Nice to meet you.
……
Nice to meet you.
Nice to meet you.^C                    # 用户按下〈Ctrl+C〉键
```

while 循环也可以是零循环，即，循环体内的命令一次也没有运行。

```
$ cat while_no_loop.sh
#!/bin/bash
i=100
while [ $i -le 3 ]
do
    echo "i="$i
    let i=$i+1
done
```

在上面的脚本中，i 的初值被赋为 100，接着判断 i 是否小于或等于 3。显然 100 大于 3，条件不成立，while 循环直接结束。运行脚本 while_no_loop.sh 将没有任何显示输出。

内置命令 false 的退出状态总是 1（失败），使用 while false 也可以组成零循环，运行下面的脚本 while_false_no_loop.sh 将没有任何显示：

```
$ cat while_false_no_loop.sh
#!/bin/bash
while false
do
    echo "Nice to meet you."
done
```

6.4　until 循环

until 循环也叫直到型循环，格式如下，do 与 done 之间为循环体：

```
until 命令
do
    命令（命令组）
done
```

或者为：

```
until 命令; do 命令（命令组）; done
```

until 先判断它后面的命令的退出状态，退出状态为非 0 时（命令不成功时，当关键字

until 后面的条件不成立时），关键字 do 之后的命令开始执行，执行到关键字 done 后，关键字 until 后面的命令再次执行，如果退出状态还是非 0（条件还是不成立），循环将继续进行，即循环体内的命令继续执行……直到 until 后面的命令执行成功（退出状态为 0、条件成立）时，循环结束。

until 循环的流程如图 6-4 所示。

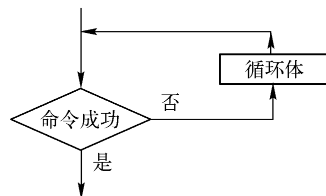


图 6-4 until 循环流程图

```
$ cat until_loop_1.sh
#!/bin/bash
i=1
until [ $i -gt 3 ]
do
    echo "i="$i
    let i=$i+1
done
```

上面的例子，先将 i 的初值赋为 1，接着判断 i 的值是否大于 3，因为 1 不大于 3，所以显示 i=1，再将 i 的值加 1；i 的值变为 2，因为 2 也不大于 3，所以显示 i=2，再将 i 的值加 1；i 的值变为 3，因为 3 仍然不大于 3，所以显示 i=3，再将 i 的值加 1；i 的值变为 4，4 大于 3，条件成立，循环结束，循环体内的命令不再执行。脚本的运行结果如下：

```
$ until_loop_1.sh
i=1
i=2
i=3
```

对比 while 和 until 循环，可以发现，while 循环就是当条件满足（关键字 while 后面的命令的退出状态为 0）时，循环体内的命令就执行；until 循环就是当条件不满足（关键字 until 后面的命令的退出状态为非 0）时，循环体内的命令就执行，直到条件满足为止。

until 循环也可以是零循环。下面的例子，i 的初值为 100，已经满足了“大于 3”的条件，所以，运行该脚本不会有任何显示输出。

```
$ cat until_no_loop.sh
#!/bin/bash
i=100
until [ $i -gt 3 ]
do
    echo "i="$i
    let i=$i+1
done
```

until 循环也可以是无穷循环。

```
$ cat until_infinite_loop.sh
#!/bin/bash
i=1
```

```
until [ $i -gt 3 ]
do
      echo "i="$i
      let i=$i-1                # 注意这里 i 的值递减，而不是递增
done
```

上面的例子，i 的初值为 1，不满足“大于 3”的条件，输出 i=1，然后 i 的值被减去 1；i 变为 0，还是不满足“大于 3”的条件，输出 i=0，然后 i 的值被减去 1；i 变为-1，仍然不满足条件……进入无穷循环。脚本运行结果如下，i 的值每次被减去 1，如果不强行终止，脚本一直显示 i 的值，直至溢出：

```
$ until_infinite_loop.sh
i=1
i=0
i=-1
i=-2
i=-3
……
i= -37687
^C                              # 用户按下〈Ctrl+C〉键
```

与 while 循环相应，使用 until false 可以构成无穷循环，使用 until true 可以构成零循环。

无穷循环运行时需要人为强行终止它。无穷循环似乎没用，其实不然。生活中就有无穷循环的应用，如取款机就是一个无穷循环的例子。假设不停电、钱永远取不完，并且没有其他故障的话，取款机是一个很好的无穷循环的例子。取款机的待机画面显示“欢迎使用，请插卡”；插卡后，机器显示相应的画面；选择取款的话，机器检查卡内余额、本次取款数额、当日累计的取款数额是否大于每日限额等……最后用户提现、取卡，取款机回到待机画面，等待下一位客户使用。下一位顾客来了后，无论是取款、修改密码、转账，还是查询余额，取款机进行着同样的循环。

可以在无穷循环中“埋设”下节要讲的 break 命令，满足某种条件时，循环终止。

6.5　用 break 和 continue 控制循环

命令 break 和 continue 可以控制循环，包括 for、while 和 until 循环。break 的作用是跳出本循环，continue 的作用是跳过本轮循环。

下面先看一个 for 循环：

```
$ cat for_loop_1_to_10.sh
#!/bin/bash
for i in 1 2 3 4 5 6 7 8 9 10
do
      echo $i;
done
```

显然，运行脚本 for_loop_1_to_10.sh 将打印输出 1 到 10。把它修改一下，在显示 i 的值之前增加判断，当 i 等于 5 时，执行 break 命令，得到下面的脚本：

```
$ cat break_for_1.sh
#!/bin/bash
for i in 1 2 3 4 5 6 7 8 9 10; do
    [ $i = 5 ] && break
    echo $i
done
```

break 的作用是终止循环，所以 i 等于 5 时循环结束，1 到 4 被打印出来，5 和后面的数字没有机会被打印。

```
$ break_for_1.sh
1
2
3
4
```

下面的脚本，在输出 i 的值之前增加了判断，当 i 等于 5 时，执行 continue 命令。

```
$ cat continue_for_1.sh
#!/bin/bash
for i in 1 2 3 4 5 6 7 8 9 10; do
    [ $i = 5 ] && continue
    echo $i
done
```

命令 continue 的作用是跳过本轮循环，直接进入下一轮循环。当 i 等于 5 时，遇到 continue 命令，则跳过 i=5 的这一轮循环，直接进入 i=6 及其之后的循环，所以除 5 之外的其他数字被打印出来。

```
$ continue_for_1.sh
1
2
3
4       # 显示 4
6       # 显示 6，没有打印 5
7
8
9
10
```

以 for 循环为例，如图 6-5 所示，break 的作用是跳出循环，continue 的作用是跳过本轮循环。

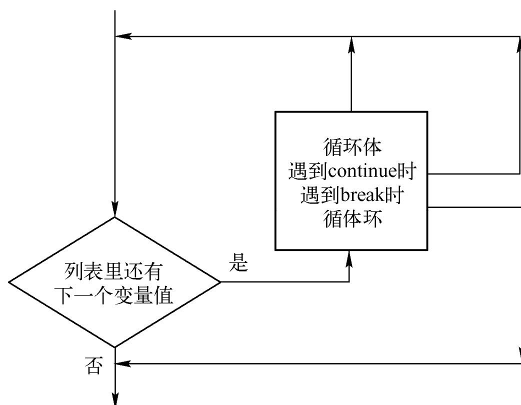


图 6-5　break 和 continue 在循环中的作用

为了更深刻地理解 break 和 continue，下面先看一个二层嵌套循环。

```
$ cat for_two_loop.sh
#!/bin/bash
for i in 1 2 3
do
    for j in a b c
    do
        echo $i#$j
    done
done
```

当 i 等于 1 时，j 从 a 循环到 c，接着 i 循环到 2，j 又从 a 循环到 c……打印输出如下：

```
$ for_two_loop.sh
1#a
1#b
1#c
2#a
2#b
2#c
3#a
3#b
3#c
```

现在，在内层循环，增加条件判断，当 j 等于 b 时，执行 break 命令。

```
$ cat break_inner_for.sh
#!/bin/bash
for i in 1 2 3
do
    for j in a b c
    do
```

```
            if [ $j = b ]; then break; fi
            echo $i#$j
        done
    done
```

当 i=1 且 j=a 时，显示 1#a；当 i=1 且 j=b 时，遇到 break，j 的循环（内层循环）终止，所以 1#b 和 1#c 没有机会被显示。i 循环到 2 和 3 时，同样道理，只有 2#a、3#a 被显示出来。

```
$ break_inner_for.sh
1#a
2#a
3#a
```

下面将 break 换为 continue，看看会发生什么。

```
$ cat continue_inner_for.sh
#!/bin/bash
for i in 1 2 3
do
    for j in a b c
    do
        if [ $j = b ]; then continue; fi
        echo $i#$j
    done
done
```

当 i=1 且 j=b 时，遇到 continue 命令，内层循环直接进入下一轮，也就是跳过 j 等于 b，开始 j 等于 c 的循环，所以，1#b 不被打印，1#a 和 1#c 被打印。当 i 等于 2 和 3 时，同样道理，2#b 和 3#b 不被打印。

```
$ continue_inner_for.sh
1#a
1#c
2#a
2#c
3#a
3#c
```

命令 break 和 continue 还可以用在 while 和 until 循环内。命令 break 和 continue 可以带参数，break N 的作用是跳出 N 层循环。看下面的例子：

```
$ cat break_two_for.sh
#!/bin/bash
for i in 1 2 3
do
    for j in a b c
```

```
        do
            if [ $j = b ]; then break 2; fi         # 注意，这里是 break 2 不是 break
            echo $i#$j
        done
    done
```

当 i=1 且 j=a 时，显示 1#a；当 i=1 且 j=b 时，执行命令 break 2，退出两层循环，对 i 的循环，对 j 的循环都结束，脚本仅仅显示 1#a 就执行完毕了：

```
$ break_two_for.sh
1#a
```

continue N 的作用是跳过其外 N 层循环的本轮循环，到达下一轮循环，看下面的例子：

```
$ cat continue_two_for.sh
#!/bin/bash
for i in 1 2 3
do
    for j in a b c
    do
        if [ $j = b ]; then continue 2; fi
        echo $i#$j
    done
done
```

当 i=1 且 j=a 时，显示 1#a；当 i=1 且 j=b 时，执行命令 continue 2，脚本有两层循环，continue 2 就是让外层循环变量 i 进入下一轮循环，开始 i=2 的循环；i=2 且 j=a 时，显示 2#a；i=2 且 j=b 时，执行命令 continue 2，让 i 进入下一轮循环，开始 i=3 的循环；i=3 且 j=a 时，显示 3#a；i=3 且 j=b 时，执行命令 continue 2，让 i 进入下一轮循环。因为 i=1、2、3 都已经循环过了，所以对 i 的循环结束，整个脚本执行完毕。因此，脚本的显示结果为 1#a，2#a 和 3#a：

```
$ continue_two_for.sh
1#a
2#a
3#a
```

6.6　用命令 shift 控制循环

前面讲特殊变量的时候，提到过位置参数。执行命令 set 10 20 30 后，位置参数的值为 $1=10，$2=20，$3=30，参数个数$#=3，利用特殊变量$#和内置命令 shift 可以对位置参数进行循环。命令 shift 用来左移位置参数，默认左移一个，就是将原来的$1 废弃，将$2 的值赋给$1，$3 的值赋给$2，等等，然后将位置参数的个数减 1。shift N 表示左移 N 个，就是将原来的$1 到$N 废弃，将${N+1}的值赋给$1，将${N+2}的值赋给$2，等等，然后将位置参数的个数减 N。看个例子就清楚了。

```
$ cat while_shift_1.sh
#!/bin/bash
set 10 20 30
while [ $# -gt 0 ]
do
    echo $1;
    shift
done
```

脚本 while_shift_1.sh，在 while 循环开始之前，$#=3，$1=10，$2=20，$3=30。接着，在循环里面打印出$1 的值（10）；然后运行 shift，位置参数左移，$#=2，$1=20，$2=30，在循环里面打印出$1 的值（20）；然后运行 shift，位置参数左移，$#=1，$1=30，在循环里面打印出$1 的值（30）；然后运行 shift，位置参数左移，$#=0，while 循环的条件——“$#大于 0”不再满足，循环结束。脚本执行结果如下：

```
$ while_shift_1.sh
10
20
30
```

$1 的值在 while 循环进行时依次变为 10、20 和 30，见表 6-1。

表 6-1　位置参数值的变化

while 循环	参数个数$#	已出局者	$1　$2　$3 的值	
第 1 轮	3	无	10　20　30	←参数输入
第 2 轮	2	10	20　30　无	←shift（左移）
第 3 轮	1	10　20	30　无　无	←shift（左移）
结束	0	10　20　30	无　无　无	←shift（左移）

在脚本 while_shift_1.sh 的基础上，将 shift 改为 shift 2，即，每次左移两个参数，得到脚本 while_shift_2.sh：

```
$ cat while_shift_2.sh
#!/bin/bash
while [ $# -gt 0 ]
do
    echo $1
    shift 2
done
```

执行该脚本，在其后面跟上 6 个参数，10 到 60，结果如下：

```
$ while_shift_2.sh 10 20 30 40 50 60
10
30
50
```

因为每次左移两个参数，所以，6 个参数只有 3 个显示出来。

在 while 循环里面增加对$#值的显示，得到脚本 while_shift_3.sh：

```
$ cat while_shift_3.sh
#!/bin/bash
while [ $# -gt 0 ]
do
      echo $1
      shift 2
      echo $#
done
```

执行该脚本，在其后面跟上 6 个参数，10 到 60，结果如下：

```
$ while_shift_3.sh 10 20 30 40 50 60
10          #  显示此时的$1 的值 10
4           #  左移 2 个参数后，参数个数为 4
30          #  显示此时的$1 的值 30
2           #  左移 2 个参数后，参数个数为 2
50          #  显示此时的$1 的值 50
0           #  左移 2 个参数后，参数个数为 0，$#大于 0 的条件不再成立，循环结束
```

6.7　选择命令 select

命令 select 一般用来显示字符界面的菜单，命令 select 的基本格式为：

```
select  变量  [ in 列表  ]
do
      命令（命令组）
done
```

见表 4-3，PS3 就是 select 命令运行时的系统提示，默认值为#?。见下例，有 5 种蔬菜供选择，脚本运行时，系统会自动按照顺序给每种蔬菜加上编号，用户输入相应的编号进行选择，脚本根据用户的选择显示用户喜爱的蔬菜名。

```
$ cat select_1.sh
#!/bin/bash
echo "please select your favorite vegetable"
select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
do
      echo "Your favorite vegetable is $vegetable."
done
```

运行该脚本，分别输入 2、5、1 后，会发现该脚本是个无穷循环，可以按〈Ctrl+C〉键退出。

```
$ select_1.sh
please select your favorite vegetable
1) beans
2) carrots
3) potatoes                              # 1)～5)为系统的显示，供用户选择
4) onions
5) rutabagas
#? 2                                     # 键盘输入 2，#?是 select 的默认提示符
Your favorite vegetable is carrots.
#? 5                                     # 键盘输入 5
Your favorite vegetable is rutabagas.
#? 1                                     # 键盘输入 1
Your favorite vegetable is beans.
#? ^C                                    # 用户按〈Ctrl+C〉键退出
```

〈Ctrl+C〉键是给运行中的脚本发送了中断信号，有“强迫”的味道。如果想“和平”一点，应该让 select 命令遇到 EOF（End Of File），〈Ctrl+D〉键是 Linux 下的 EOF。对于上面的脚本，用户可以按〈Ctrl+D〉键“和平地”退出。用〈Ctrl+C〉键退出还是用〈Ctrl+D〉键退出，从表面看，似乎没有区别，实际上完全不一样。假设脚本 select_1.sh 中 select 命令关键字 done 的后面还有其他命令，按〈Ctrl+C〉键时，整个脚本被中断了，done 后面的命令不能运行；按〈Ctrl+D〉键时，只是 select 命令正常退出了，后面的命令可照常运行。

脚本 select_1.sh 中的 select 是无穷循环，但通常不会让一个菜单选择成为无穷循环，也不要求用户都知道按〈Ctrl+D〉键可以正常退出一个菜单选择，而是在菜单选择项中增加一个 quit。在脚本里，一般菜单选择项 quit 对应命令 break，用命令 break 跳出循环。

修改脚本 select_1.sh，先将 PS3 的赋值为一个较为合适的提示，然后在供选择的蔬菜当中增加 quit，得到脚本 select_2.sh：

```
$ cat select_2.sh
#!/bin/bash
PS3="please select your favorite vegetable: "      # 设置 select 运行时的系统提示
select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas" "quit"
do
    if [ $vegetable = quit ]
    then
        break                  # 用户可选择 quit 来退出，而不必按〈Ctrl+C〉键退出
    fi
    echo "Your favorite vegetable is $vegetable."
done
```

运行 select_2.sh，可见系统提示变为 please select your favorite vegetable:，不再是#?。

任意输入 1～5 之间的数字，会显示相应的蔬菜名。如想退出脚本，输入 6，即选择 quit，则脚本中的命令 break 得到执行，select 菜单选择命令结束，整个脚本结束，而不用按〈Ctrl+C〉键。

```
$ select_2.sh
```

```
1) beans             3) potatoes           5) rutabagas
2) carrots           4) onions             6) quit
please select your favorite vegetable: 1
Your favorite vegetable is beans.
please select your favorite vegetable: 4
Your favorite vegetable is onions.
please select your favorite vegetable: 3
Your favorite vegetable is potatoes.
please select your favorite vegetable: 6
```

内置变量 COLUMNS 用于设定 select 命令的菜单的宽度，即列数，默认为 80。内置变量 LINES 用于设定 select 命令的在垂直方向上的行数，默认为 24。

与 for 循环一样，select 基本格式中的“in 列表”不是必须的，如果没有“in 列表”，那么“select 变量”相当于“select 变量 in "$@"”，即脚本参数将成为菜单选项。例如，脚本 select_3.sh 中的 select 没有“in 列表”部分。

```
$ cat select_3.sh
#!/bin/bash
COLUMNS=10
LINES=6
PS3="please select your favorite vegetable: "
select vegetable
do
    if [ $vegetable = quit ]
    then
        break
    fi
    echo "Your favorite vegetable is $vegetable."
done
```

运行 select_3.sh，参数为 beans carrots potatoes onions rutabagas quit，效果与脚本 select_2.sh 的相同：

```
$ select_3.sh beans carrots potatoes onions rutabagas quit
1) beans
2) carrots
3) potatoes
4) onions
5) rutabagas
6) quit
please select your favorite vegetable: 1
Your favorite vegetable is beans.
please select your favorite vegetable: 2
Your favorite vegetable is carrots.
please select your favorite vegetable: 5
Your favorite vegetable is rutabagas.
```

```
please select your favorite vegetable: 6
```

从前面的例子可知，select 命令在运行时，会自动给列表里面的每个变量值加上编号。这个编号存储在内置变量 REPLY 中。见脚本 select_REPLY.sh，供选择的蔬菜的编号（No.）通过内置变量$REPLY 显示。

```
$ cat select_REPLY.sh
#!/bin/bash
PS3="please select your favorite vegetable: "
select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas" "quit"
do
    if [ $vegetable = quit ]
    then
        echo "You quit now"      # 执行 break 之前，显示 You quit now
        break
    fi
    echo "Your favorite vegetable is $vegetable. The No. is $REPLY"
done
```

运行脚本 select_REPLY.sh，键盘输入 5 和 2 时，相应的蔬菜名字和编号显示出来，在键盘输入 6 时，脚本退出：

```
$ select_REPLY.sh
1) beans          3) potatoes          5) rutabagas
2) carrots        4) onions            6) quit
please select your favorite vegetable: 5
Your favorite vegetable is rutabagas. The No. is 5
please select your favorite vegetable: 2
Your favorite vegetable is carrots. The No. is 2
please select your favorite vegetable: 6
You quit now
```

命令 select 常常与命令 case 配合。脚本 select_case.sh 根据选择的蔬菜，显示相应的价格。在脚本 select_case.sh 中，假设 beans 和 potatoes 的价格都是 2.58，carrots 和 rutabagas 的价格都是 3.65，onions 的价格是 4.12。case 命令中的*)部分是对不在列表里面的键盘输入进行处理。

```
$ cat select_case.sh
#!/bin/bash
PS3="please select your favorite vegetable: "
select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
do
    case $vegetable in
      beans | potatoes)
        echo "you select $vegetable, the price is 2.58"
        break
```

```
            ;;
        carrots | rutabagas)
            echo "you select $vegetable, the price is 3.65"
            break
            ;;
        onions)
            echo "you select $vegetable, the price is 4.12"
            break
            ;;
        *)
            echo "your selection is $REPLY, it is not correct"
            ;;
    esac
done
```

运行 select_case.sh，键盘输入 1～5 间的数字，则显示相应的蔬菜名和价格，输入 1～5 之外的数字，则提示输入错误并要求用户重新输入：

```
$ select_case.sh
1) beans
2) carrots
3) potatoes
4) onions
5) rutabagas
please select your favorite vegetable: 5
you select rutabagas, the price is 3.65
$ select_case.sh                              # 再运行一次
1) beans
2) carrots
3) potatoes
4) onions
5) rutabagas
please select your favorite vegetable: 8      # 键盘输入 8
your selection is 8, it is not correct        # 提示输入错误
please select your favorite vegetable: 1
you select beans, the price is 2.58
```

前面已经解释了内置变量 REPLY 的作用。把脚本 select_case.sh 中 case 命令的变量由 vegetable 换为 REPLY，beans | potatoes) 要相应地变为 1 | 3)，得到脚本 select_case_REPLY.sh，它与脚本 select_case.sh 的作用完全一致。

```
$ cat select_case_REPLY.sh
#!/bin/bash
PS3="please select your favorite vegetable: "
select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
do
    case $REPLY in
```

```
            1 | 3 )
              echo "you select $vegetable, the price is 2.58"
              break
              ;;
            2 | 5 )
              echo "you select $vegetable, the price is 3.65"
              break
              ;;
            4 )
              echo "you select $vegetable, the price is 4.12"
              break
              ;;
            *)
              echo "your selection is $REPLY, it is not correct"
              ;;
        esac
    done
```

运行 select_case_REPLY.sh，键盘输入 1～5 间的数字，显示相应的蔬菜名和价格，输入 1～5 之外的数字，提示输入错误并要求用户重新输入：

```
$ select_case_REPLY.sh
1) beans
2) carrots
3) potatoes
4) onions
5) rutabagas
please select your favorite vegetable: 4          #  键盘输入 4
you select onions, the price is 4.12              # 4 对应 onions
$ select_case_REPLY.sh                            #  再运行一次
1) beans
2) carrots
3) potatoes
4) onions
5) rutabagas
please select your favorite vegetable: 7          #  输入 1～5 之外的数
your selection is 7, it is not correct            #  提示输入不正确
please select your favorite vegetable: 1          #  输入 1
you select beans, the price is 2.58               # 1 对应 beans
```

6.8 循环命令与 I/O 重定向及管道的配合

while 循环可以使用输入重定向，对输入文件的每一行进行处理。例如，下面的文件是一个名字列表：

```
$ cat name_list.txt
Tom
Jack
Harry
Merry
```

脚本 while_input_redirection.sh 的 while 循环的第一行为 while read name，最后一行为 done < name_list.txt，表示将文件 name_list.txt 的每一行读入到变量 name 中：

```
$ cat while_input_redirection.sh
#!/bin/bash
while read name
do
    echo "Good morning, $name."
done < name_list.txt
```

运行脚本，while 循环依次对文件中的每个人进行了早安问候：

```
$ while_input_redirection.sh
Good morning, Tom.
Good morning, Jack.
Good morning, Harry.
Good morning, Merry.
```

下面举一个 for 循环的输出通过管道线传给 Linux 命令的例子。下面的脚本中的 for 循环可以显示几个字母：

```
$ cat for_sort_1.sh
#!/bin/bash
for i in c b n m p d
do
    echo $i
done
```

运行脚本 for_sort_1.sh，显示了这几个字母。但是，字母没有按照字母表顺序排列：

```
$ for_sort_1.sh
c
b
n
m
p
d
```

在脚本 for_sort_1.sh 的后面加管道线和排序命令 sort，字母就按照字母表顺序排列了：

```
$ for_sort_1.sh | sort
b
```

```
c
d
m
n
p
```

可以将管道线和排序命令 sort 直接放在脚本里面关键字 done 的后面，得到脚本 for_sort_2.sh：

```
$ cat for_sort_2.sh
#!/bin/bash
for i in c b n m p d
do
    echo $i
done | sort
```

运行脚本 for_sort_2.sh，显示了按照字母表顺序排列好的字母：

```
$ for_sort_2.sh
b
c
d
m
n
p
```

下面的脚本 while_cat_redirection.sh 中的 while 循环处在一条 cat 命令及管道线的后面，cat 输出的每一行被关键字 while 后面的 read 命令读取并存入变量 name 中。while 循环的最后一行为 done > tmp_name$$。$$是脚本运行时的进程 ID（见表 4-2），它的值是不固定的，假设$$=1234，while 循环中的 echo 命令的显示就会重定向到临时文件 tmp_name1234。

```
$ cat while_cat_redirection.sh
#!/bin/bash
cat $1 | while read name
do
    echo "Good morning, $name."
done > tmp_name$$
cp tmp_name$$ $1
```

运行脚本 while_cat_redirection.sh，参数为前面刚刚提到过的文件 name_list.txt：

```
$ while_cat_redirection.sh name_list.txt
```

于是脚本的参数$1=name_list.txt。name_list.txt 的内容由 cat 命令输出，通过管道传递给 while 循环。while 循环接到 name_list.txt 第一行的内容后，读入到变量 name 中，echo 命令显示 Good morning, Tom。同理，读入第二行后，显示显示 Good morning, Jack……直到文件

内容读入完毕。while 循环结束后，所有的显示并未出现在屏幕，而是重定向到临时文件。查看一下临时文件是否存在：

```
$ ls tmp_name*
tmp_name29861
```

从临时文件的名字可知脚本运行时的进程 ID 是 29861。查看临时文件的内容：

```
$ cat tmp_name29861
Good morning, Tom.
Good morning, Jack.
Good morning, Harry.
Good morning, Merry.
```

临时文件 tmp_name29861 就是在原 name_list.txt 文件的基础上，每行前面增加了 Good morning。

脚本 while_cat_redirection.sh 的最后一行为 cp tmp_name$$ $1，就是将临时文件复制为 name_list.txt。复制之后，name_list.txt 的内容与临时文件一致，每行的前面增加了"早安"，查看一下新的 name_list.txt：

```
$ cat name_list.txt
Good morning, Tom.
Good morning, Jack.
Good morning, Harry.
Good morning, Merry.
```

在对文件 file.txt 的每一行进行相同或有规律的处理时，常用：

```
while read line
do
    对$line 的处理命令
done < file.txt
```

也常用：

```
cat file.txt | while read line
do
    对$line 的处理命令
done
```

6.9　脚本的选项与参数

6.9.1　命令 shift

当一个脚本需要带选项和参数时，可以用循环命令与 shift 命令配合，对位置参数也就

是脚本的选项和参数进行处理。

例如脚本 shift_options.sh 的功能是，带选项-a 时，显示 Good afternoon；带选项-m 时，显示 Good morning；带选项-n 时，显示 Good night；带选项-w 及相应的参数数值时，显示本周是今年第几周，并显示 Have a nice weekend。

```
$ cat shift_options.sh
#!/bin/bash
while [ $# -ge 1 ]
do
    opt=$1
    case $opt in
        -a) echo "Good afternoon"
            shift
            ;;
        -m) echo "Good morning"
            shift
            ;;
        -n) echo "Good night"
            shift
            ;;
        -w) shift
            week_number=$1
            if [ "$week_number" == "" ]
            then
                echo "after -w, there must be a week number"
                exit 1;
            fi
            echo "This is week $week_number, Have a nice weekend"
            shift
            ;;
         *) echo "Usage is : $0 [-a] [-m] [-n] [-w number]"
            exit 1
            ;;
    esac
done
```

当只有一个选项-a 时，脚本参数个数$#=1，只有一轮 while 循环，显示 Good afternoon：

```
$ shift_options.sh -a
Good afternoon
```

当有两个选项-m 和-n 时，$#=2，有两轮 while 循环，显示 Good morning 和 Good night：

```
$ shift_options.sh -m -n
```

```
Good morning
Good night
```

当带上选项-w 及参数 28 时，$#=2，但只有一轮 while 循环，显示“本周是第 28 周，周末愉快”：

```
$ shift_options.sh -w 28
This is week 28, Have a nice weekend
```

当输入的选项是-a，-m，-n，-w 之外的东西，脚本中的 case 命令将匹配*，脚本将显示其用法（Usage）：

```
$ shift_options.sh x
Usage is : ./shift_options.sh [-a] [-m] [-n] [-w number]
$ shift_options.sh -q
Usage is : ./shift_options.sh [-a] [-m] [-n] [-w number]
```

当只带上选项-w，忘了其参数时，脚本显示提示：

```
$ shift_options.sh -w
after -w, there must be a week number
```

脚本中与选项-w 有关的部分，可以加强，如判断 week_number 必须是整数，范围须在 1～52 之间（因为一年只有 52 周）等，但这方面的内容不是本节所要讨论的重点。

用循环命令与 shift 命令配合的方式来处理选项和参数有缺点。一个缺点是，选项无法像 Linux 命令那样合并。例如，运行 shift_options.sh -mn 时，效果与 shift_options.sh -m -n 的不一致：

```
$ shift_options.sh -mn
Usage is : ./shift_options.sh [-a] [-m] [-n] [-w number]
```

另外一个缺点是，当选项和参数比较多时，脚本中的 shift 命令就会很多，使得脚本不容易阅读和维护。

6.9.2　命令 getopts

使用 getopts 命令可以丢掉 6.9.1 节脚本中的 shift 命令。getopts 命令的一般格式为：

```
getopts optstring variable
```

optstring 是一个包括字母和冒号的字符串（字母一定有，冒号不一定有）。getopts 检查脚本的位置参数中是否有以减号开头的字母并且该字母出现在 optstring 当中，若是有，这样的字母被认为是脚本中的选项，并且该选项被存入变量 variable 中。在 optstring 中，字母的后面有冒号时，该选项将包含参数；字母的后面没有冒号时，该选项没有参数。看下面的脚本及运行情况，再读前面这句话就清楚了。

```
$ cat getopts_1.sh
```

```
#!/bin/bash
while getopts amnw: OPTION
do
    case $OPTION in
      a) echo Good afternoon
         ;;
      m) echo Good morning
         ;;
      n) echo Good night
         ;;
      w) week_number=$OPTARG
         echo "This is week $week_number, Have a nice weekend"
         ;;
      *) echo "Usage is : $0 [-amn] [-w number]"
         exit 1;
         ;;
    esac
done
```

当运行 getopts_1.sh -a 时，脚本中的 getopts 命令发现 a 出现在 amnw:这个字符串当中，于是将 a 存入变量 OPTION 中；在 case 命令中，OPTION 与 a 匹配，所以，显示 Good afternoon：

```
$ getopts_1.sh -a
Good afternoon
```

脚本 getopts_1.sh 比上一小节的脚本少了 shift 命令，而且像 Linux 命令一样，多个选项可以合并。运行 getopts_1.sh -mn 时，有两轮 while 循环，变量 OPTION 分别取值为 m 和 n，显示两句话，效果与 getopts_1.sh -m -n 相同：

```
$ getopts_1.sh -mn
Good morning
Good night
```

运行 getopts_1.sh -w 18 时，getopts 命令发现 w 出现在 amnw:这个字符串当中，并且 w 的后面有冒号，这表明：正常的话，-w 后面有参数。内置变量 OPTARG 用来存放参数的值，所以，18 被存入了 OPTARG 中。仔细读脚本，不难理解下面命令的输出结果：

```
$ getopts_1.sh -w 18
This is week 18, Have a nice weekend
```

当输入不正确的选项时，系统将提示出错。同时，不正确的选项使得 case 命令中的 $OPTION 与*匹配，于是脚本显示了用法：

```
$ getopts_1.sh -q
./getopts_1.sh: illegal option -- q                # 系统提示出错，不当选项
```

```
Usage is : ./getopts_1.sh [-amn] [-w number]                # 脚本显示的用法
```

当输入正确的选项，但是应该带参数而没有带时，系统也提示出错：

```
$ getopts_1.sh -w
./getopts_1.sh: option requires an argument -- w            # 系统提示，需要参数（argument）
Usage is : ./getopts_1.sh [-amn] [-w number]
```

选项字符串 optstring 中的字母出现的次序无关紧要，脚本 getopts_1.sh 中的 optstring 为 amnw:，写为 aw:nm，效果不变。变次序时，要注意，w 的后面有冒号，每个冒号要“紧跟”自己的字母。

如果选项字符串 optstring 以冒号开头，例如，将脚本中的 amnw:变为:amnw:，那么，输入不当选项（illegal option）和缺乏参数（option requires an argument）时，系统将不提示出错。没有了系统错误提示的话，脚本的编写者就需要对选项和参数严格检查。实际上，脚本 getopts_1.sh 中 case 命令对模式*的处理就是对不正确的选项的处理。

选项字符串 optstring 以冒号开头时，虽然系统不提示出错，但 getopts 会区分不当选项和缺少参数这两种错误。输入不当选项时，variable 被设成?，缺少参数时，variable 被设成:。选项字符串 optstring 不以冒号开头时，getopts 不区分这两种错误，遇到这两种错误时，variable 都被设成?（提示，这里的 variable 指的是本小节一开始给出的 getopts 命令格式中的 variable）。

脚本 getopts_1.sh 中的$0 在显示脚本用法时得到的是./getopts_1.sh。./表示当前目录，并不“碍事”，如果不希望出现./可以使用命令 basename。

综上所述，修改 getopts_1.sh 的几处，得到 getopts_2.sh。其中，amnw:变为:amnw:，为了表明字母出现的次序无关紧要，:amnw:又变为:aw:nm；*)变为'?'|':')；$0 变为$(basename $0)，或者变为`basename $0`也可以。

```
$ cat getopts_2.sh
#!/bin/bash
while getopts :aw:nm OPTION
do
    case $OPTION in
      a) echo Good afternoon
         ;;
      m) echo Good morning
         ;;
      n) echo Good night
         ;;
      w) week_number=$OPTARG
         echo "This is week $week_number, Have a nice weekend"
         ;;
      '?'|':') echo "Usage is : $(basename $0) [-amn] [-w number]"
         exit 1;
         ;;
    esac
```

```
done
```

输入一个错误选项-Y：

```
$ getopts_2.sh -Y
Usage is : getopts_2.sh [-amn] [-w number]
```

输入选项-w，后面忘记带数字了：

```
$ getopts_2.sh -w
Usage is : getopts_2.sh [-amn] [-w number]
```

输入合并在一起的选项-na，运行正常：

```
$ getopts_2.sh -na
Good night
Good afternoon
```

输入选项-w 及参数 20，运行正常：

```
$ getopts_2.sh -w 20
This is week 20, Have a nice weekend
```

与 getopts 命令相关的还有一个内置变量 OPTIND，见表 4-3，它的初始值为 1，在循环中，它的值为脚本的下一个参数的索引。为了理解 OPTIND，看脚本 getopts_3.sh，在 case 命令的后面增加了对变量 OPTIND 的值的显示命令 echo "OPTIND=$OPTIND"：

```
$ cat getopts_3.sh
#!/bin/bash
while getopts aw:mn OPTION
do
    case $OPTION in
      a) echo Good afternoon
         ;;
      m) echo Good morning
         ;;
      n) echo Good night
         ;;
      w) echo "This is week $OPTARG"
         ;;
      *) echo input error
         exit 1;
         ;;
    esac
    echo "OPTIND=$OPTIND"
done
```

运行 getopts_3.sh -a -w 30 -n -m 时，可以认为有 6 个位置参数，依次为-a、-w、30、-n、

-m、无。明明是 5 个，为什么偏说是 6 个、最后一个是“无”呢？看完本小节就明白了。-a 对应地输出 Good afternoon，下一个为-w（第 2 个位置参数），所以处理完-a 后 OPTIND=2；-w 30 对应地输出 This is week 30，下一个为-n（第 4 个位置参数），所以处理完-w 30 后 OPTIND=4；-n 对应地输出 Good night，下一个为-m（第 5 个位置参数），所以处理完-n 后 OPTIND=5；-m 对应地输出 Good morning，下一个为无（第 6 个位置参数），所以最后显示 OPTIND=6：

```
$ getopts_3.sh -a -w 30 -n -m
Good afternoon
OPTIND=2
This is week 30
OPTIND=4
Good night
OPTIND=5
Good morning
OPTIND=6
```

运行 getopts_3.sh -anm -w 30 时，可以认为有 4 个位置参数，依次为-anm、-w、30、无。-anm 是-a -n -m 的简写形式，但对 shell 而言，-anm 是一个位置参数。-a 对应地输出 Good afternoon，下一个为 n，它处在第 1 个位置参数之中，所以处理完-a 后 OPTIND=1；n 对应地输出 Good night，下一个为 m，m 仍处在第 1 个位置参数之中，所以处理完-n 后 OPTIND=1；m 对应地输出 Good morning，下一个为-w，它为第 2 个位置参数，所以处理完-m 后 OPTIND=2；-w 30 对应地输出 This is week 30，下一个为无，它为第 4 个位置参数，所以最后显示 OPTIND=4：

```
$ getopts_3.sh -anm -w 30
Good afternoon
OPTIND=1
Good night
OPTIND=1
Good morning
OPTIND=2
This is week 30
OPTIND=4
```

第 7 章　函　　数

Bash 函数与其他语言（如 C 语言）函数的意义是一致的。可以把完成某一特定功能的任务交给一个函数来完成，这符合模块化的思想。如果一个较长的程序里面有重复的代码，可以将重复的代码提取出来，变为一个函数，在相应的地方调用这个函数即可。这样做，有利于程序的维护，也可减少代码冗余。总之，函数用来模块化程序，提高程序的维护效率。

7.1　函数定义

Bash 通常使用以下两种格式定义函数，学习过 C 语言的人非常眼熟：

```
function 函数名
{
    命令（命令组）
}
函数名()
{
    命令（命令组）
}
```

定义函数时，如果使用了关键字 function，函数名字后面不用跟小括号；如果没有使用关键字 function，函数名字后面要加一对小括号。

函数必须先定义，后使用。下面这个脚本里的函数 greeting 用来打印一句问候语：

```
$ cat fun_greet_1.sh
#!/bin/bash
function greeting
{
    echo "Good morning."
}
greeting                        # 调用函数 greeting
```

脚本的最后一行为函数名，用于调用该函数。运行脚本，可以打印出这句问候语：

```
$ fun_greet_1.sh
Good morning.
```

当函数很短时，函数的定义可以在一行之内完成，例如：

```
function test_fun { echo "Good morning"; }
```

在一行之内定义函数时，左大括号的两边要有空格，右大括号与最后一条命令之间要有分号，否则会出现语法错误。

Bash 还接受第三种函数定义的格式，就是既有关键字 function，函数名后又跟有小括号：

```
function 函数名()
{
    命令（命令组）
}
```

喜欢使用这种格式的人不少，因为它的可读性最好。

7.2　给函数传递参数

前面例子中的函数是一个没有参数的函数。可以通过位置参数给函数传递参数值，看下面的例子：

```
$ cat fun_greet_2.sh
#!/bin/bash
function greeting
{
    echo "Good morning, $1, $2, and $3."
}
greeting Mike Jack Mary   # 调用函数 greeting，函数参数是 Mike，Jack 和 Mary
```

greeting 后面的 Mike、Jack 和 Mary 分别传给了位置参数$1、$2 和$3，脚本 fun_greet_2.sh 运行结果如下：

```
$ fun_greet_2.sh
Good morning, Mike, Jack, and Mary.
```

上面的例子要事先知道传给函数几个参数。有时传给函数的参数的个数不是固定的，这时，可以利用特殊变量$#和内置命令 shift。在第 6 章已经讲过$#和 shift 的使用，再看一个用在函数上的例子：

```
$ cat fun_shift.sh
#!/bin/bash
function greeting
{
    while [ $# -gt 0 ]
    do
        echo "Good morning, $1."
        shift
    done
}
greeting Mike Jack Mary      # 调用函数 greeting，函数参数是 Mike，Jack 和 Mary
```

脚本的最后一行调用了函数 greeting，函数的参数个数$#=3，参数分别为，$1=Mike，$2=Jack，$3=Mary。有了 6.6 节的基础，很容易明白脚本的执行结果：

```
$ fun_shift.sh
Good morning, Mike.
Good morning, Jack.
Good morning, Mary.
```

7.3 函数的局部与全局变量

在一个函数内可以定义变量，变量名可以与主程序的变量同名。如果同名，它们会相互影响，因为函数内的变量默认同主程序的变量一样，都是全局变量。

```
$ cat fun_global_1.sh
#!/bin/bash
function global_b
{
    b=200
    echo "in function global_b, b=$b"
}
b=180
echo "in Main script, b=$b"
global_b
echo "now, in Main script, b=$b"
```

上面的脚本，主程序里将 b 赋值为 180 并显示其值；然后调用函数 global_b，在函数内部，b 被赋值为 200 并显示其值；接着在主程序里再显示 b 的值。看看执行结果：

```
$ fun_global_1.sh
in Main script, b=180              #  显示出刚刚赋的值 180
in function global_b, b=200        #  在函数 global_b 内，给 b 赋值并显示其值 200
now, in Main script, b=200         #  函数调用完毕，回到主程序，b 的值变为 200
```

因为函数里面的变量 b 和主程序的变量 b，实际上是同一个 b——全局变量 b，所以在函数内对一个全局变量赋值，就是对主程序的同一个变量赋值，结果主程序最后一句显示 b 的值为 200。

如果用内置命令 local 将函数里面的变量定义为局部变量，看看会是什么情况。

```
$ cat fun_local_1.sh
#!/bin/bash
function local_b
{
    local b=200
    echo "in function local_b, b=$b"
}
```

```
b=180
echo "in Main script, b=$b"
local_b
echo "now, in Main script, b=$b"
```

函数 local_b 里面定义的 b 是局部变量，作用范围只在该函数内部。主程序的变量 b 与函数里面的变量 b 虽然名字相同但不是同一个 b。不难理解脚本的执行结果：

```
$ fun_local_1.sh
in Main script, b=180           # 显示出刚刚赋的值 180
in function local_b, b=200      # 在函数 local_b 内，给局部变量 b 赋值并显示其值
now, in Main script, b=180      # 函数调用完毕，回到主程序，b 的值为 180，不变
```

在函数内定义变量，前面加 declare 时，与加 local 效果一致，也是将变量定义为局部变量，看脚本 fun_declare_local.sh 及其运行结果就明白了：

```
$ cat fun_declare_local.sh
#!/bin/bash
function local_b
{
    declare b=206                    # 在函数内用内置命令 declare 定义局部变量
    echo "in function local_b, b=$b"
}
b=180
echo "in Main script, b=$b"
local_b
echo "now, in Main script, b=$b"
```

运行 fun_declare_local.sh：

```
$ fun_declare_local.sh
in Main script, b=180
in function local_b, b=206
now, in Main script, b=180
```

在函数内用 declare -g 定义变量时，变量为全局（global）变量，看下面的脚本及其运行结果就明白了：

```
$ cat fun_declare_global.sh
#!/bin/bash
function global_b
{
    declare -g b=206              # 在函数内用 declare -g 命令定义全局变量
    echo "in function global_b, b=$b"
}
b=180
echo "in Main script, b=$b"
```

```
global_b
echo "now, in Main script, b=$b"
```

运行 fun_declare_global.sh：

```
$ fun_declare_global.sh
in Main script, b=180
in function global_b, b=206
now, in Main script, b=206
```

在一个函数内，直接定义一个变量，或者用 declare -g 定义变量，都是将变量定义为全局变量。显然，用 declare -g，可读性好一些。在一个函数内，定义一个变量时，前面加上 local 或者 declare，都是将变量定义为局部变量。显然，用 local，可读性好一些。

7.4 当前的函数名 FUNCNAME

通过 Bash 的内置变量 FUNCNAME，可以知道正在被调用的函数的名字。下面的例子，先定义函数 fun_1 和 fun_2，在这两个函数的内部都显示内置变量 FUNCNAME 的值，然后依次调用 fun_1 和 fun_2，最后，在主程序里再显示内置变量 FUNCNAME 的值：

```
$ cat fun_FUNCNAME.sh
#!/bin/bash
function fun_1
{
    echo "at this moment, function name is $FUNCNAME"
}
function fun_2
{
    echo "at this moment, function name is $FUNCNAME"
}
fun_1                #调用函数 fun_1
fun_2                #调用函数 fun_2
echo "at this moment, function name is $FUNCNAME"     #在主程序里 FUNCNAME 的值为空
```

调用函数 fun_1 和 fun_2 时，FUNCNAME 的值分别是 fun_1 和 fun_2，在主程序（不在某个函数内）显示 FUNCNAME 的值时，FUNCNAME 的值为空，运行结果如下：

```
$ fun_FUNCNAME.sh
at this moment, function name is fun_1     # 脚本运行至函数 fun_1 时显示的内容
at this moment, function name is fun_2     # 脚本运行至函数 fun_2 时显示的内容
at this moment, function name is           # 脚本运行在主程序显示的内容
```

以上对 FUNCNAME 的讲解没有错，但并不完整，实际上 FUNCNAME 是一个数组，记录调用链上所有的函数名。${FUNCNAME[0]}代表当前正在执行的函数的名字，${FUNCNAME[1]}代表调用函数${FUNCNAME[0]}的函数的名字，以此类推，数组最后元

素为主程序 main。前面提到的所谓的变量 FUNCNAME 的值，实际上是数组 FUNCNAME 的首个元素值。

看下面的例子。在函数 fun_2 中，调用了函数 fun_1，在主程序中，调用了函数 fun_2，在两个函数中都显示了数组 FUNCNAME 的值：

```
$ cat fun_FUNCNAME_2.sh
#!/bin/bash
function fun_1
{
    echo "Now, function name is $FUNCNAME"
    declare -p FUNCNAME
}
function fun_2
{
    echo "Now, function name is $FUNCNAME"
    declare -p FUNCNAME
    fun_1
}
#main script start
fun_2
```

运行脚本 fun_FUNCNAME_2.sh，当函数 fun_2 执行时，数组 FUNCNAME 的首个元素为 fun_2，下一个元素为 main，表示主程序调用了函数 fun_2；当函数 fun_1 执行时，数组 FUNCNAME 的首个元素为 fun_1，下一个元素为 fun_2，最后元素为 main，表示主程序调用了函数 fun_2，fun_2 调用了函数 fun_1：

```
$ fun_FUNCNAME_2.sh
Now, function name is fun_2
declare -a FUNCNAME='([0]="fun_2" [1]="main")'
Now, function name is fun_1
declare -a FUNCNAME='([0]="fun_1" [1]="fun_2" [2]="main")'
```

工作当中，有些脚本很长，包含很多函数，函数可以相互调用。有时在调试脚本的时候，脚本运行到哪个函数里了，可能会不清楚。利用 FUNCNAME，就可以知道脚本调用函数的“轨迹”。

7.5　在命令行执行函数

脚本 fun_export.sh 中定义了函数 sum_of_two_integer，该函数用于计算两个整数的和。

```
$ cat fun_export.sh
#!/bin/bash
sum_of_two_integer()
{
    local sum
```

```
    let sum=$1+$2
    echo $sum
}
```

用 source 命令执行脚本 fun_export.sh：

```
$ source fun_export.sh
```

上面的 source 命令执行之后，当前 shell 里面就有了这个函数的定义，可以在命令行调用该函数，就像执行命令一样。下面计算 10 与 35 的和：

```
$ sum_of_two_integer 10 35
45          # 得到了正确的结果 45
```

运行 sum_of_two_integer 10 35 时，在函数 sum_of_two_integer 的内部，$1=10，$2=35，sum=10+35=45，显示结果 45。

可以像调用其他命令那样，用$(函数名)或$(函数名 参数…)，或者用反引号——`函数名`，或`函数名 参数…`，将函数的运行结果赋给变量，下面将 50 与 100 的和赋给 a：

```
$ a=$(sum_of_two_integer 50 100)
$ echo $a
150         # 得到了正确的结果 150
```

用反引号格式，将 60 与 25 的和赋给 a：

```
$ a=`sum_of_two_integer 60 25`
$ echo $a
85          # 得到了正确的结果 85
```

7.6　查看当前 shell 的函数定义

要查看当前 shell 的某个函数的定义，可用如下命令，其中选项-f 是 function 的意思：

declare -f 函数名

下面的命令可以查看函数 sum_of_two_integer 的定义：

```
$ declare -f sum_of_two_integer
sum_of_two_integer ()
{
    local sum;
    let sum=$1+$2;
    echo $sum
}
```

上面命令中的选项-f 可以换为-F，命令 declare -F <函数名>的输出仅仅是函数名，没有函数体，例如：

```
$ declare -F sum_of_two_integer
sum_of_two_integer
```

运行命令 declare -f <函数名>或者命令 declare -F <函数名>时，如果函数在当前 shell 中未定义，或者函数名输入错误，declare 命令将没有输出。

运行 declare -F，当前 shell 的所有函数的名字显示出来；运行 declare -f，当前 shell 的所有函数名及其函数体定义显示出来。

7.7　选项-f 与函数的导出、清除与只读设置

函数可以像变量一样导出，格式如下：

export -f 函数名

或者

declare -fx 函数名

下面将函数 sum_of_two_integer 导出，导出后，在子 shell 中可以使用它。

```
$ export -f sum_of_two_integer                # 或者运行 declare -fx 函数名
```

要查看当前 shell 里面所有导出的函数，可运行命令 export -fp，或 declare -fx：

```
$ export -fp                                  # 或者用命令 declare -fx
sum_of_two_integer ()
{
    local sum;
    let sum=$1+$2;
    echo $sum
}
declare -fx sum_of_two_integer
```

取消当前 shell 里面某个函数的定义，命令为：

```
unset -f 函数名
```

下面取消当前 shell 里面函数 sum_of_two_integer 的定义，再调用该函数，计算 10 与 20 的和，得到了 command not found 的信息，因为该函数的定义已经不存在了：

```
$ unset -f sum_of_two_integer
$ sum_of_two_integer 10 20
sum_of_two_integer: command not found
```

像普通变量一样，函数也可以是只读的。用 readonly -f 加函数名或者 declare -rf 加函数名可以使函数变为只读的。例如：

```
$ function hello { echo Hello, morning; }
$ readonly -f hello
```

不能取消只读函数的定义，也不能重新定义一个只读函数。如：

```
$ unset -f hello
bash: unset: hello: cannot unset: readonly function
$ function hello { echo Good afternoon; }
bash: hello: readonly function
```

7.8　返回命令 return

内置命令 return 用于从函数中返回。前面关于函数的例子中并没有看到 return，因为 return 命令在函数中不是必须的，函数被调用时，函数内的命令执行完成后自然会返回到调用它的地方（一般是返回到主程序）。如果某函数中有 return 命令，执行到 return 时就返回。如果 return 在函数中不是最后一条命令，那么 return 后面的其他命令不再执行。

看下面的例子，函数里面有两条打印命令，中间是返回命令。

```
$ cat fun_return_1.sh
#!/bin/bash
function return_1
{
    echo "return_1_1"
    return
    echo "return_1_2"              # 这一行命令不会执行，因为它在命令 return 的后面
}
return_1                           # 调用函数 return_1
```

运行脚本，函数 return_1 内的第一条 echo 命令被执行，然后执行 return，返回到主程序，第二条 echo 命令不被执行。

```
$ fun_return_1.sh
return_1_1
```

注意 return 命令不能直接用在脚本的主程序里，看下面的例子：

```
$ cat fun_return_2.sh
#!/bin/bash
echo "return_2_1"
return
echo "return_2_2"
```

如果直接运行脚本，会遇到错误提示：

```
$ fun_return_2.sh
return_2_1
./fun_return_2.sh: line 3: return: can only `return' from a function or sourced script
return_2_2
```

但用 source 命令或者点命令来运行该脚本，就没问题：

```
$ source fun_return_2.sh
return_2_1
```

先执行脚本 fun_return_2.sh 的第一条 echo 命令，显示 return_2_1，然后执行 return，脚本结束，第二条 echo 命令不被执行。

return 命令可以带参数，格式为 return N。一个函数通过 return 返回到主程序，该函数的返回状态值（退出状态）决定于 return 前一条命令的退出状态；一个函数通过 return N 返回到主程序，那么该函数的返回状态值就是 N。

脚本 fun_no_return.sh 中定义了两个函数，都没有命令 return。函数 fun_date 中是一条 date 命令，函数 fun_datee 中是一条不存在的 datee 命令：

```
$ cat fun_no_return.sh
#!/bin/bash
function fun_date()
{
    date
}
function fun_datee()
{
    datee
}
```

运行 source 命令，使得这两个函数在当前 shell 中生效：

```
$ source fun_no_return.sh
```

调用函数 fun_date，其实就是执行 date 命令，执行成功，函数 fun_date 的退出状态为 0：

```
$ fun_date
Mon Mar 11 10:59:47 EDT 2013
$ echo $?
0
```

调用函数 fun_datee，其实就是执行一条不存在的命令 datee，执行失败，函数 fun_datee 的退出状态为 127，也就是执行不存在命令的退出状态：

```
$ fun_datee
datee: command not found
$ echo $?
127
```

脚本 fun_return_number.sh 中两个函数都包括命令 return：

```
$ cat fun_return_number.sh
#!/bin/bash
```

```
function fun_date()
{
    date
    return 25
}
function fun_datee()
{
    datee
    return 0
}
```

运行 source 命令，使得这两个函数在当前 shell 中生效：

```
$ source fun_return_number.sh
```

调用函数 fun_date，其实就是执行 date 命令，执行成功，而函数 fun_date 的退出状态为 25，不再是 0，因为函数中有命令 return 25：

```
$ fun_date
Mon Mar 11 11:09:43 EDT 2013
$ echo $?
25
```

调用函数 fun_datee，其实就是执行一条不存在的命令 datee，执行失败，而函数 fun_datee 的退出状态为 0，不再是 127，因为函数中有命令 return 0：

```
$ fun_datee
datee: command not found
$ echo $?
0
```

在函数中，也可以使用命令 exit，执行 exit 后整个脚本结束运行。而在函数中执行 return 的话，将回到脚本中调用该函数的地方，而不是让脚本结束运行。这是 exit 与 return 的区别。

在 7.4 节 fun_FUNCNAME.sh 的基础上，在第一个函数中加入命令 exit，得到如下脚本：

```
$ cat fun_exit.sh
#!/bin/bash
function fun_1
{
    echo "at this moment, function name is $FUNCNAME"
    exit
}
function fun_2
{
    echo "at this moment, function name is $FUNCNAME"
```

```
}
fun_1               # 调用函数 fun_1
fun_2               # 调用函数 fun_2
```

执行脚本 fun_exit.sh，只有第一个函数被调用了，因为第一个函数中的命令 exit 运行后，整个脚本结束运行，第二个函数没有被调用的机会：

```
$ fun_exit.sh
at this moment, function name is fun_1    # 只有函数 fun_1 被调用了
```

7.9　递归函数

像 C 和 Pascal 语言一样，Bash 支持函数的递归定义。递归函数，就是直接或者间接地调用自身的函数。

最典型的递归是阶乘的运算，整数 N 的阶乘为：N!=1×2×3×…×(N-1)×N=(N-1)!×N，即 N 的阶乘等于 1 乘以 2 再乘以 3……再乘以 N，也等于 N-1 的阶乘再乘以 N。脚本 fun_recursion.sh 中定义了计算阶乘的递归函数 factorial。

```
$ cat fun_recursion.sh
#!/bin/bash
factorial()
{
    local M K N fac
    let N=$1                            # 为便于理解，将参数保存为 N
    if [[ ($N == 1) || ($N == 0) ]]     # 或者写为 if [ $N -eq 1 -o $N -eq 0 ]
    then
        echo 1                          # 当 N=1 或者 N=0 时，N 的阶乘等于 1
    else
       let M=$N-1                       # M 的值为 N-1
       K=$(factorial $M)                # 函数调用自身 factorial，将 N-1 的阶乘存入 K
       let fac=$K*$N                    # N-1 的阶乘（K），乘以 N，得到 N 的阶乘，存入 fac
       echo $fac                        # 显示 fac 的值，即 N 的阶乘值
    fi
}

factorial 4                             # 调用函数，计算 4 的阶乘
factorial 10                            # 计算 10 的阶乘
factorial 15                            # 计算 15 的阶乘
factorial 20                            # 计算 20 的阶乘
```

运行脚本 fun_recursion.sh，分别得到了 4 的阶乘、10 的阶乘、15 的阶乘和 20 的阶乘：

```
$ fun_recursion.sh
24                                      # 4!=24
3628800                                 # 10!=3628800
```

```
1307674368000                  # 15!=1307674368000
2432902008176640000            # 20!=2432902008176640000
```

4 的阶乘为 24，看一看递归函数是如何计算的：4 的阶乘等于 4 乘以 3 的阶乘，3 的阶乘等于 3 乘以 2 的阶乘，2 的阶乘等于 2 乘以 1 的阶乘，而 1 的阶乘等于 1，到达了递归出口；于是，2 的阶乘等于 2 乘以 1、结果为 2；3 的阶乘等于 3 乘以 2 的阶乘、结果为 6；4 的阶乘等于 4 乘以 3 的阶乘，结果为 24。

如果说一说递归函数是如何计算 20 的阶乘的话，那么上面一段内容将变得很长。用递归算法解题通常显得很简洁，但递归算法的运行效率较低，速度慢，而且占用系统资源较多，一般不提倡使用递归。在 Bash 脚本中，递归函数的使用不多见。

第 8 章　正则表达式与文本处理

简易的文本编辑工具，如记事本和写字板，只能进行比较简单的查找和替换。功能强大的文本编辑工具，如 UltraEdit 和 Notepad++，支持正则表达式，即具有按照某种模式（如通配符）进行查找和替换（包括批量替换）的功能。Linux 下的很多工具具备较强的文本处理能力，其中一些工具支持正则表达式，本章将介绍常用文本处理工具的基本功能。

8.1　文件名替换

假设当前目录下扩展名为.txt 的文件有很多，如果一个一个地删除这些文件，会比较慢，运行命令 rm *.txt 则可以一下子把它们都删除。这条命令使用了文件名替换，*.txt 代表所有名字以.txt 结尾的文件。

8.1.1　多字符替换*

假设某个目录下面有如下 9 个文件：

```
$ ls
a    aa    ab    ab.sh    b    bb    x1.txt    y2.txt    z3.txt
```

在命令行里，*会被替换为当前目录下的所有的文件（不包括隐藏文件）：

```
$ ls *
a    aa    ab    ab.sh    b    bb    x1.txt    y2.txt    z3.txt
```

用 echo *也可以显示当前目录下的所有文件：

```
$ echo *
a aa ab ab.sh b bb x1.txt y2.txt z3.txt
```

运行 cat *，将显示当前目录下所有文件的内容。
a*代表所有以名字 a 开头的文件名：

```
$ ls a*
a    aa    ab    ab.sh
```

*2*代表所有包含数字 2 的文件名：

```
$ ls *2*
y2.txt
```

*.*代表所有包含小数点的文件名：

```
$ ls *.*
ab.sh    x1.txt    y2.txt    z3.txt
```

当在命令行里的*的两边带引号（单、双引号均可）或者*前面带反斜杠时，*就是它自己，不再代表当前目录下的所有的文件。下面的 touch 命令可以产生一个新文件，名字就叫*：

```
$ touch "*"                    # 注意*的两边有引号
$ ls "*"
*
```

而 touch *（注意*的两边没有引号）的作用是将当前目录下所有文件的时间戳更新。要删除这个名字特殊的文件，可以用命令 rm "*"（注意*的两边有引号）：

```
$ rm "*"
$ ls "*"                       # 删除它之后，再查看它，则不存在了
ls: cannot access *: No such file or directory
```

通过 ls "*"的显示结果，可知这个名字特殊的文件被删除了。不要用 rm *（*的两边没有引号）删除它，如果运行 rm *，当前目录下所有的文件将被删除。

8.1.2 单字符替换?

问号?代表一个字符，ls ?将显示名字为一个字符的文件名：

```
$ ls ?
a    b
```

??表示两个字符，ls ??将显示名字为两个字符的文件名：

```
$ ls ??
aa    ab    bb
```

ls a?将显示名字以 a 开头，a 后面为一个字符的文件名：

```
$ ls a?
aa    ab
```

ls x?.*将显示名字以 x 开头，后跟一个字符和小数点，小数点后面有任意个字符的文件名：

```
$ ls x?.*
x1.txt
```

ls *.??将显示名字包含小数点，小数点后面有两个字符的文件名：

```
$ ls *.??
ab.sh
```

8.1.3　范围替换[]与[!]

单字符的替换除了用?之外，还可以使用中括号。中括号里面可以列出具体的字符，字符之间是或的关系。例如，[amz]*表示名字以 a，或以 m，或以 z 开头的文件名：

```
$ ls [amz]*
a    aa    ab    ab.sh    z3.txt
```

中括号里面还可以给出字符的范围，例如[a-z]表示任意一个小写字母，[A-Z]表示任意一个大写字母，[A-Za-z]表示任意一个字母，[b-mx-z] 表示 b 到 m、x 到 z 之间的任意一个小写字母。ls [a-z]将显示名字为小写字母的单字符文件名：

```
$ ls [a-z]
a    b
```

ls [a-k][a-k]将显示名字包含 a 到 k 之间的字母的双字符文件名：

```
$ ls [a-k][a-k]
aa    ab    bb
```

[a-k]*表示以 a 到 k 之间的字母开头的文件名：

```
$ ls [a-k]*
a    aa    ab    ab.sh    b    bb
```

*[0-9]*表示包含一个阿拉伯数字的文件名：

```
$ ls *[0-9]*
x1.txt    y2.txt    z3.txt
```

*[2-9]*表示包含一个 2 到 9 的阿拉伯数字的文件名：

```
$ ls *[2-9]*
y2.txt    z3.txt
```

中括号里面的第一个字符为!时，表示取反匹配。例如，[!0-9]表示任意一个非数字的字符，[!a-z]表示任意一个非小写字母的字符，[!az]*表示不以字母 a 和 z 开头的文件名：

```
$ ls [!az]*
b    bb    x1.txt    y2.txt
```

[!p-z]*表示不以 p 到 z 之间的字母开头的文件名：

```
$ ls [!p-z]*
a    aa    ab    ab.sh    b    bb
```

注意，用中括号指定范围的时候，前面字符的序号不应大于后面的字符，例如[z-a]是不合理的：

```
$ ls [z-a]*
ls: cannot access [z-a]*: No such file or directory
```

内置变量 GLOBIGNORE（见表 4-3）的值会影响文件名的匹配。本来*代表目录下所有的文件：

```
$ ls *
a    aa    ab    ab.sh    b    bb    x1.txt    y2.txt    z3.txt
```

把 GLOBIGNORE 的值设置如下，意思是，名字以 a 和以 b 开头的文件，被忽略匹配：

```
$ GLOBIGNORE=a*:b*
```

现在再运行 ls *，名字以 a 和以 b 开头的文件不显示了：

```
$ ls *
x1.txt    y2.txt    z3.txt
```

8.2　正则表达式与 grep

正则表达式是用来描述或者匹配一系列符合某种规则的字符串，是一个描述一组字符串的模板。使用正则表达式的目的几乎都是为了进行文本处理。

8.2.1　过滤器 grep

命令 grep（global search regular expression and print out the line，全面搜索正则表达式并把行打印出来）是一个过滤器，它可以对一个或者多个文件按照某种字符串模式（pattern）搜索出相应的内容，它的一般格式为：

```
grep [选项]  查找模式  文件名  [文件名…]
```

例如，文件 name_list.txt 有两列，分别是名字和性别：

```
$ cat name_list.txt
Tom          boy
Jack         boy
Harry        boy
Merry        girl
```

列出文件中的男孩：

```
$ grep boy name_list.txt
Tom          boy
Jack         boy
Harry        boy
```

列出文件中的女孩：

```
$ grep girl name_list.txt
Merry       girl
```

grep 命令的输入也可以来自管道。例如，要查询都有谁登录了 Linux 服务器，可以用 who 命令：

```
$ who
shiqdong   pts/0      Sep 20 09:15 (:1)
ssha11x    pts/10     Aug 27 19:11 (dog30:30.0)
manwatn    pts/72     Sep 11 13:51 (doclln07:4459.0)
ssha11x    pts/76     Aug 28 08:39 (dog30:30.0)
shiqdong   pts/16     Sep 21 04:16 (:1)
ccadmin    pts/60     Jul 25 05:07 (docsxbora09.example.com)
ewntte     pts/102    Sep 19 17:31 (dosxpalau41.example.com:73.0)
ewntte     pts/105    Sep 20 14:19 (dosxpalau41.example.com:73.0)
ewntte     pts/106    Sep 19 17:53 (dosxpalau41.example.com:73.0)
ewntte     pts/108    Sep 19 18:04 (dosxpalau41.example.com:73.0)
ewntte     pts/109    Sep 19 18:35 (dosxpalau41.example.com:73.0)
shiqdong   pts/20     Aug 30 07:59 (:1)
shiqdong   pts/35     Aug 30 06:02 (:1)
shiqdong   pts/6      Sep 20 09:40 (:1)
skajbmer   pts/37     Aug 31 14:01 (dog32:55.0)
shiqdong   pts/36     Sep 20 10:06 (:1)
shiqdong   pts/27     Aug 30 06:01 (:1)
shiqdong   pts/2      Sep 20 09:32 (:1)
werkler    pts/112    Sep 20 10:44 (dosxdodo44.example.com:46.0)
ewntte     pts/93     Sep 19 17:26 (dosxpalau41.example.com:73.0)
ewntte     pts/78     Sep 19 17:26 (dosxpalau41.example.com:73.0)
ewntte     pts/98     Sep 19 17:29 (dosxpalau41.example.com:73.0)
werkler    pts/99     Sep 20 10:44 (dosxdodo44.example.com:46.0)
werkler    pts/113    Sep 20 10:44 (dosxdodo44.example.com:46.0)
werkler    pts/114    Sep 20 10:44 (dosxdodo44.example.com:46.0)
ewntte     pts/120    Sep 20 14:41 (dosxpalau41.example.com:73.0)
bkoxhevn   pts/124    Sep 20 15:30 (:pts/123:S.0)
```

而要查询一下某个账户如 werkler 登录了没有，在哪里登录的，在长长的名单里用肉眼查找很困难。此时使用 grep 命令对 who 命令的输出过滤一下就知道了：

```
$ who | grep werkler
werkler   pts/112    Sep 20 10:44 (dosxdodo44.example.com:46.0)
werkler   pts/99     Sep 20 10:44 (dosxdodo44.example.com:46.0)
werkler   pts/113    Sep 20 10:44 (dosxdodo44.example.com:46.0)
werkler   pts/114    Sep 20 10:44 (dosxdodo44.example.com:46.0)
```

上面的命令是将命令 who 的输出过滤一下，只显示包含字符串 werkler 的行。如果只记得账户的前 2 个字符为 we、后 2 个字符为 er，可以这样查找：

```
$ who | grep we.*er
```

```
werkler   pts/112       Sep 20 10:44 (dosxdodo44.example.com:46.0)
werkler   pts/99        Sep 20 10:44 (dosxdodo44.example.com:46.0)
werkler   pts/113       Sep 20 10:44 (dosxdodo44.example.com:46.0)
werkler   pts/114       Sep 20 10:44 (dosxdodo44.example.com:46.0)
```

其中“.”表示任何单个字符，“*”表示匹配零次或者多次，“.*”就表示零个或任意多个字符。常用的模式匹配字符及含义见表 8-1。

表 8-1 模式匹配

字 符	模 式 含 义
.	匹配任何单个字符
*	匹配零次或者多次
?	匹配零次或者一次（扩展）
+	匹配一次或者多次（扩展）
\{N\}	精确匹配 N 次（扩展的用{N}）
\{N,\}	匹配 N 次或者更多次（扩展的用{N,}）
\{N,M\}	匹配至少 N 次，至多 M 次（扩展的用{N,M}）
a|b|c	匹配 a 或 b 或 c（扩展）
( )	分组符号，如 read(able|dress)匹配 readable 或者 readdress（扩展）
[x-y]	范围，如[0-9]匹配数字，[a-z]匹配小写字母，[A-Z0-9]匹配大写字母或数字
[ ]	匹配一组字符中的一个，如[anz8]表示 a,n,z,8 中的任一字符
^	匹配行始
$	匹配行末
[^]	列表范围取反，如[^0-9]表示非数字，[^anz8]表示 a,n,z,8 之外的任意字符
\	转义（escapes）一个特殊字符，屏蔽某些字符的特殊含义，表示原来的意思
\b	单词定界符
\<	词首定界符
\>	词尾定界符
\w	匹配字母数字下画线（单词字符），即[A-Za-z0-9_]
\W	\w 取反，非单词字符，即[^A-Za-z0-9_]

下面举例说明表 8-1 中相关的内容。注意下面的文件一共 4 行，其中第 3 行是空行：

```
$ cat test.txt
map story
big Map 12345

story book
```

因为“.”代表任意一个字符，所以 grep .会过滤出非空行：

```
$ grep . test.txt
map story
big Map 12345
```

```
story book
```

过滤出包含 map 的行：

```
$ grep map test.txt
map story
```

grep 的选项-i 或者--ignore-case 用来忽略字母大小写，过滤出包含 map（忽略大小写）的行：

```
$ grep -i map test.txt
map story
big Map 12345
```

表 8-1 中的^和$叫做锚。锚指明了正则表达式在一行文本中要匹配的位置，如\b，\<和\>也是锚。下面过滤出以 map 开头的行：

```
$ grep ^map test.txt
map story
```

过滤出以 book 结尾的行：

```
$ grep book$ test.txt
story book
```

^匹配行始，$匹配行尾，那么^$匹配空行：

```
$ grep ^$ test.txt
[空行]
```

grep 的选项-v 用来取反（invert）匹配。命令 grep -v ^$的作用就是匹配非空行：

```
$ grep -v ^$ test.txt
map story
big Map 12345
story book
```

假设文件 test.txt 现在变为如下内容：

```
$ cat test.txt
map story
12345
story 6789 book
```

显示包含阿拉伯数字的行：

```
$ grep [0-9] test.txt
12345
story 6789 book
```

显示包含非阿拉伯数字的行：

```
$ grep [^0-9] test.txt
map story
story 6789 book
```

不显示包含阿拉伯数字的行，其他行都显示：

```
$ grep -v [0-9] test.txt
map story
```

正则表达式里面的“^”和 grep 命令的选项-v，都有取反的功能，但是具体作用是不一样的。细细比较并体会前面的两个例子就清楚了。

下面显示出包含 2 个连续字母 o 的行：

```
$ grep 'o\{2\}' test.txt
story 6789 book
```

下面显示出包含至少 5 个连续的阿拉伯数字的行：

```
$ grep '[0-9]\{5,\}' test.txt
12345
```

如果想在当前目录下的所有文件中搜索 book，运行 grep book *，*被替换为当前目录下的所有文件：

```
$ grep book *
Linux.txt:There are many Linux books.
test.txt:story 6789 book
```

上面的输出表示文件 Linux.txt 和 test.txt 包含 book，相应的整行内容也显示出来了。如果只想知道哪些文件包含 book，用选项-l：

```
$ grep -l book *
Linux.txt
test.txt
```

选项-n 的作用是显示匹配行在文件中的行号：

```
$ grep -n book *
Linux.txt:3:There are many Linux books.
test.txt:3:story 6789 book
```

上面的输出表示文件 Linux.txt 的第 3 行和 test.txt 的第 3 行包含 book，相应的整行内容也显示出来了。

在匹配.和*等特殊字符本身的时候需要注意，要用反斜杠屏蔽其特殊的含义。先看一个文件：

```
$ cat star.txt
It's a txt file.
5 * 9 = 45
********************
Bey-bey
```

要想过滤出包含*的行，直接运行 grep * star.txt 是不行的，因为，在命令行中，*被替换为当前目录下的所有文件，在*前面加反斜杠（或者在*的两边加单引号或双引号）才可以过滤出包含*的行：

```
$ grep \* star.txt
5 * 9 = 45
********************
```

grep 命令的选项-c 或者--count 的作用是显示匹配的次数。文件 star.txt 中包含*的行有两行，所以下面命令的结果为 2：

```
$ grep -c \* star.txt
2
```

过滤出包含 . 的行：

```
$ grep '\.' star.txt
It's a txt file.
```

直接运行 grep . star.txt 是不行的，感兴趣的话，可以试试。

下面是关于单词定界符的例子，文件 man.txt 的内容及相关例子如下：

```
$ cat man.txt
This man is my friend
That manxyz read a book
Your xyzman eat fish
telephone
the
take
$ cat man.txt | grep '\<man'          # 过滤出包含以 man 开头的单词的行
This man is my friend
That manxyz read a book
$ cat man.txt | grep '\<man\>'        # 过滤出包含单词 man 的行
This man is my friend
$ cat man.txt | grep 'man\>'          # 过滤出包含以 man 结尾的单词的行
This man is my friend
Your xyzman eat fish
$ grep '\bxyz' man.txt                # 过滤出包含以 xyz 开头的单词的行
Your xyzman eat fish
$ grep 'xyz\b' man.txt                # 过滤出包含以 xyz 结尾的单词的行
That manxyz read a book
```

过滤出包含以 t 开头、以 e 结尾、中间有两个字符的单词的行：

```
$ grep '\<t..e\>' man.txt
take
```

过滤出包含以 t 开头、以 e 结尾、中间有任意个字符的单词的行：

```
$ grep '\<t.*e\>' man.txt
telephone
the
take
```

8.2.2　扩展的 egrep

表 8-1 中的一些行标注了“扩展”。命令 egrep 是 grep 的扩展，支持更多元字符。例如，[0-9]?表示零个或一个阿拉伯数字。在字符串 abc2ef 中包含一个数字，可以用[0-9]？匹配：

```
$ echo abc2ef | egrep '[0-9]?'
abc2ef
```

如果把上面命令中的 egrep 换为 grep 就过滤不出来。带选项-E 或者--extended-regexp 的 grep 命令相当于 egrep 命令，把上面命令中的 egrep 换为 grep -E，效果是相同的。

字符串 abcd 恰好有 4 个小写字母：

```
$ echo abcd | egrep '[a-z]{4}'
abcd
```

上面的命令不需要在大括号左边加反斜杠了，如果用 grep 命令，需要加反斜杠，即 grep '[a-z]\{4\}'。

下面的 egrep 命令可过滤出 readable 或者 readability，而 grep 没有这样的功能：

```
$ echo "it's readable" | egrep 'read(able|ability)'
it's readable
```

平常提到的正则表达式简称为 BRE（Basic Regular Expressions），扩展正则表达式简称为 ERE（Extended Regular Expressions）。

8.2.3　POSIX 字符类

这里给出另外一种用于指定匹配字符范围的方法，就是 POSIX 字符类，见表 8-2。

表 8-2　POSIX 字符类

字　符　类	含　　义
[:alnum:]	字母和数字，等同于[A-Za-z0-9]
[:alpha:]	字母，等同于[A-Za-z]
[:blank:]	空格或制表符（Tab）

（续）

字　符　类	含　　义
[:cntrl:]	控制字符
[:digit:]	十进制数字，等同于[0-9]
[:graph:]	可打印的图形字符，即 ASCII 码值 33～126 的字符
[:lower:]	小写字母，等同于[a-z]
[:print:]	可打印字符，即 ASCII 码值 32～126 的字符，比[:graph:]多一个空格字符
[:punct:]	标点符号
[:space:]	空格、制表符、竖向制表符、换行、回车等
[:upper:]	大写字母，等同于[A-Z]
[:xdigit:]	十六进制数字，等同于[0-9A-Fa-f]

在应用的时候，表 8-2 中的字符类需要用双中括号括起来。下面看几个相关的例子。posix.txt 是包含各种字符的文本文件：

```
$ cat posix.txt
678
abc
ABCD
GOOD AFTERNOON
"Pig123"
,:!
```

过滤出有标点符号的行：

```
$ grep [[:punct:]] posix.txt
"Pig123"
,:!
```

过滤出有空格的行：

```
$ grep [[:blank:]] posix.txt
GOOD AFTERNOON
```

过滤出有十六进制字符的行：

```
$ grep [[:xdigit:]] posix.txt
678
abc
ABCD
GOOD AFTERNOON
"Pig123"
```

过滤出有大写字母的行：

```
$ grep [[:upper:]] posix.txt
ABCD
```

```
GOOD AFTERNOON
"Pig123"
```

过滤出有连续 5 个或者 5 个以上大写字母的行：

```
$ grep '[[:upper:]]\{5,\}' posix.txt     # 或者运行 egrep '[[:upper:]]{5,}' posix.txt
GOOD AFTERNOON
```

过滤出包含一个大写字母紧接着一个小写字母的行：

```
$ grep '[[:upper:]][[:lower:]]' posix.txt
"Pig123"
```

8.2.4 Bash 扩展模式匹配

Bash 的内置命令 shopt 用来设置当前 shell 的选项（options）。选项-s 用来打开（set）某个选项，运行命令 shopt -s extglob 可以打开扩展模式匹配。开启 extglob 之后，有五种模式匹配可被识别，见表 8-3，表中的 pattern-list 是一个或者多个模式，多个模式用“|”隔开。

表 8-3　扩展模式

模　　式	含　　义
?(pattern-list)	匹配所给模式零次或一次
*(pattern-list)	匹配所给模式零次或多次
+(pattern-list)	匹配所给模式一次或多次
@(pattern-list)	匹配所给模式仅一次
!(pattern-list)	不匹配所给的模式

看几个例子。先给变量 name 赋值为 Tommy，再打开扩展模式匹配：

```
$ name=Tommy
$ shopt -s extglob
```

[Tt]表示 T 或者 t，+(m)表示一个或者多个 m，所以 Tommy 与[Tt]o+(m)y 匹配，测试一下：

```
$ [[ $name == [Tt]o+(m)y ]]
$ echo $?
0                    # 匹配，所以退出状态为 0
```

@(m)表示一个 m，所以 Tommy 与[Tt]o@(m)y 不匹配，测试一下：

```
$ [[ $name == [Tt]o@(m)y ]]
$ echo $?
1                    # 不匹配，所以退出状态为 1
```

将 name 的值改为 Tomy，@(m)表示一个 m，?(m)表示一个 m 或者没有 m，下面两个都

成功匹配：

```
$ name=Tomy
$ [[ $name == [Tt]o@(m)y ]]
$ echo $?
0
$ [[ $name == [Tt]o?(m)y ]]
$ echo $?
0
```

在一个空目录下面产生几个空文件：

```
$ touch 1.txt 2.gif 3.pdf 4.xml
$ ls
1.txt   2.gif   3.pdf   4.xml
```

@(txt|xml)表示 txt 或者 xml，要删除名字结尾为 txt 或 xml 的文件，可以用下面的命令，它与 rm -rf *txt *xml 等效：

```
$ rm -rf *@(txt|xml)
```

查看一下，可知删除成功：

```
$ ls
2.gif   3.pdf
```

删除不以 pdf 结尾的文件：

```
$ rm -rf !(*pdf)
```

查看一下，可知删除成功，只剩下以 pdf 结尾的文件了：

```
$ ls
3.pdf
```

8.3　剪取内容命令 cut

cut 命令可根据不同的选项，提取每个输入行指定的相应内容，并显示出来。cut 命令对输入的每一行进行同样的处理，常用格式为：

```
cut  选项  文件名
```

它的最常用的选项为-c，-c 后边紧跟字符序列清单（characters list）。先看一封简短的 email：

```
$ cat Linux.txt
Hello Jack,
```

```
I like Linux, it's very interesting.
There are many Linux books.
Best regards,
Mike
```

显示每一行的第一个字符，用选项-c1：

```
$ cut -c1 Linux.txt
H
I
T
B
M
```

显示每一行的第 1 个字符和第 3 到 5 个字符，用选项-c1,3-5：

```
$ cut -c1,3-5 Linux.txt
Hllo
Ilik
Tere
Bst
Mke
```

显示每一行的第 1 个、第 3 到 5 个和第 10 个到行尾的字符，用选项-c1,3-5,10-，其中 10-表示第 10 个到行尾的字符：

```
$ cut -c1,3-5,10- Linux.txt
Hllok,
Iliknux, it's very interesting.
Tere many Linux books.
Bst rds,
Mke
```

显示每一行的前 3 个字符，用选项-c1-3，因为从行首开始，可以简写为-c-3：

```
$ cut -c-3 Linux.txt
Hel
I l
The
Bes
Mik
```

下面介绍 cut 命令另外两个常用的选项-d 和-f。先介绍-f，选项-f 用来指定域（field list）。下面的文件有 3 个域（3 列单词），域之间的分隔符为〈Tab〉键：

```
$ cat tab.txt
aaa bbb xxx
ccc ddd yyy
```

因为〈Tab〉键是不可打印的字符，如何知道域分隔符为〈Tab〉键的呢？用命令 sed -n l 可以显示，命令 sed 将在后面介绍。下面的显示结果中，\t 表示〈Tab〉键，$表示行尾：

```
$ sed -n l tab.txt
aaa\tbbb\txxx$
ccc\tddd\tyyy$
```

显示文件的第 2 个域，用选项-f2：

```
$ cut -f2 tab.txt
bbb
ddd
```

显示文件的第 1 个和第 3 个域，用选项-f1,3：

```
$ cut -f1,3 tab.txt
aaa xxx
ccc yyy
```

如果文件的域分隔符不是〈Tab〉键，可以使用 cut 命令的选项-d 指定域分隔符（field delimiter）。先查看某 Linux 系统的文件/etc/passwd，显然，该文件的域分隔符为冒号：

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
```

下面命令中的-d:指定了域分隔符为冒号，选项-f1,7 的作用是让第 1 个和第 7 个域显示出来：

```
$ cut -d: -f1,7 /etc/passwd
root:/bin/bash
daemon:/bin/sh
bin:/bin/sh
sys:/bin/sh
sync:/bin/sync
```

8.4　合并相应行的命令 paste

paste 命令的作用与 cut 相反，它不是把一行剪开，而是将多行“粘”在一起。例如，有如下 3 个文件，分别记录了几个人的名字、相应的年龄、相应的专业：

```
$ cat name.txt                # name.txt 是记录名字的文件
Mike
```

```
Marry
Jack
$ cat age.txt                # age.txt 是记录年龄的文件
25
28
23
$ cat major.txt              # major.txt 是记录专业的文件
History
English
Math
```

如下的 paste 命令会将这 3 个文件对应的行合并为一行，显示在标准输出。合并后的域之间的分隔符默认为〈Tab〉键：

```
$ paste name.txt age.txt major.txt
Mike	25	History
Marry	28	English
Jack	23	Math
```

paste 命令的选项-d 用来指定域分隔符。下面命令中的选项-d#使得域分隔符变为#：

```
$ paste -d# name.txt age.txt major.txt
Mike#25#History
Marry#28# English
Jack#23# Math
```

paste 命令的选项-s 的作用是，以每个文件为一个处理单元，将文件内容在一行输出，多行合为一行，默认以〈Tab〉键作为域分隔符，如：

```
$ paste -s name.txt
Mike	Marry	Jack
```

下面的命令依次将 3 个文件所有的行分别合为一行：

```
$ paste -s name.txt age.txt major.txt
Mike	Marry	Jack
25	28	23
History	English	Math
```

下面举个 cut 命令与 paste 命令配合的例子，它们的输入均来自管道，使用前面提到过的文件/etc/passwd。如果想知道在该文件中的账号名，需要把第一列提取出来：

```
$ cat /etc/passwd | cut -d: -f1
root
daemon
bin
sys
sync
```

账号名得到了，让它们在一行显示：

```
$ cat /etc/passwd | cut -d: -f1 | paste -s
root        daemon        bin     sys     sync
```

现在，得到了文件/etc/passwd 当中的账号名，但账号名之间为〈Tab〉键，如果希望账号名之间为空格，在 paste 命令中加入选项-d' '（两个单引号之间为一个空格）：

```
$ cat /etc/passwd | cut -d: -f1 | paste -d' ' -s
root daemon bin sys sync
```

8.5 转换或删除字符命令 tr

tr 命令主要用来转换（translate）字符，一般格式为：

```
tr from-characters to-characters
```

即：

```
tr 转换前的字符 转换后的字符
```

使用输入重定向，将 8.3 节提到过的文件 Linux.txt 中的所有的字母 i 转换为 x：

```
$ tr i x < Linux.txt
Hello Jack,
I lxke Lxnux, xt's very xnterestxng.
There are many Lxnux books.
Best regards,
Mxke
```

使用输入重定向，将 Linux.txt 中的所有的小写字符转换为大写字符：

```
$ tr [a-z] [A-Z] < Linux.txt
HELLO JACK,
I LIKE LINUX, IT'S VERY INTERESTING.
THERE ARE MANY LINUX BOOKS.
BEST REGARDS,
MIKE
```

在 tr 命令中，大写字母与小写字母除了可以用[A-Z]和[a-z]表示之外，还可以用[:upper:]和[:lower:]表示。tr 命令支持 POSIX 字符类，见 8.2.3 节。使用管道，将 Linux.txt 中的所有的大写字符转换为小写字符：

```
$ cat Linux.txt | tr [:upper:] [:lower:]
hello jack,
i like linux, it's very interesting.
there are many linux books.
best regards,
```

```
mike
```

将数字转换为字母 X：

```
$ echo "I am 28 years old" | tr '[[:digit:]]' X
I am XX years old
```

tr 命令支持八进制表示的字符，见表 8-4。

表 8-4　tr 命令支持的字符

速　记　符	八 进 制 值	含　　义
\a	\007	响铃
\b	\010	退格
\f	\014	换页
\n	\012	新行
\r	\015	回车
\t	\011	〈Tab〉键（横向制表）
\v	\013	竖向制表（垂直跳格）
\\		反斜线本身
\NNN		ASCII 码为一到三位八进制数对应的字符

用 echo 命令显示一句话，包含 4 个单词，单词之间为一个空格：

```
$ echo "I study bash script."
I study bash script.
```

把空格都转换为换行符，则每个单词都单独占据一行显示：

```
$ echo "I study bash script." | tr ' ' '\n'
I
study
bash
script.
```

用八进制\012 代替\n，效果是一样的：

```
$ echo "I study bash script." | tr ' ' '\012'
I
study
bash
script.
```

tr 命令的选项-s 用来压缩重复的字符（squeeze-repeats），将重复的字符变为一个。下面的 echo 命令显示了有重复字符的一句话，通过命令 tr -s 去掉小写字母中的重复字符：

```
$ echo "I ssstudyyyyy bashhhhh scrippppt." | tr -s [a-z]
```

```
I study bash script.
```

因为重复的字符都是小写字母，下面的命令没有起到去掉重复字符的作用：

```
$ echo "I ssstudyyyyy bashhhhh scrippppt." | tr -s [A-Z]
I ssstudyyyyy bashhhhh scrippppt.
```

tr 命令的选项-d 用来删除（delete）字符。下面的命令显示了 Linux.txt 中的所有空格被去掉之后的样子：

```
$ cat Linux.txt | tr -d ' '
HelloJack,
IlikeLinux,it'sveryinteresting.
TherearemanyLinuxbooks.
Bestregards,
Mike
```

Linux.txt 中的所有大写字母被去掉之后的样子：

```
$ cat Linux.txt | tr -d [A-Z]
ello ack,
  like inux, it's very interesting.
here are many inux books.
est regards,
ike
```

8.6　排序命令 sort

sort 命令的作用是排序。

前面提到过，想知道文件/etc/passwd 中的账号名，用如下命令：

```
$ cat /etc/passwd | cut -d: -f1
root
daemon
bin
sys
sync
```

账号名没有按照字典顺序升序或者降序排列。而 sort 命令默认进行升序排列：

```
$ cat /etc/passwd | cut -d: -f1 | sort
bin
daemon
root
sync
sys
```

选项-r 或者--reverse 用来颠倒排列顺序：

```
$ cat /etc/passwd | cut -d: -f1 | sort -r
sys
sync
root
daemon
bin
```

选项-u 用来去掉重复的行，保证每一行的内容是唯一的（unique）。先查看文件/etc/passwd 中各个账号的默认 shell，即文件的第 7 列：

```
$ cat /etc/passwd | cut -d: -f7
/bin/bash
/bin/sh
/bin/sh
/bin/sh
/bin/sync
```

可见，/bin/sh 显示了 3 次，用命令 sort -u 可以对其排序，并去掉重复的行：

```
$ cat /etc/passwd | cut -d: -f7 | sort -u
/bin/bash
/bin/sh
/bin/sync
```

选项-R 和--random-sort 的作用是随机排序。下面对/etc/passwd 文件中的账号名进行几次随机排序，每次结果都不一样：

```
$ cat /etc/passwd | cut -d: -f1 | sort -R
sys
bin
sync
root
daemon
$ cat /etc/passwd | cut -d: -f1 | sort -R
bin
daemon
sys
sync
root
$ cat /etc/passwd | cut -d: -f1 | sort -R
bin
root
sync
sys
daemon
```

在讲解 sort 命令的选项-n 之前，先看一个纯数值的文件：

```
$ cat number.txt
8
2
19
12
1
15
```

下面对文件 number.txt 排序：

```
$ sort number.txt
1
12
15
19
2
8
```

sort 命令对纯数值文件的内容排序时，将数值按照普通字符串对待，对其按照字典顺序进行升序排序。因为 1 在 2 的前面，12、15、19 等都在 2 的前面，就像 a 在 b 的前面，单词 about、ahead、arrive 等在字典里都排在 b 的前面。

选项-n 的作用是让 sort 命令按照数值大小排序（numeric-sort）：

```
$ sort -n number.txt
1
2
8
12
15
19
```

前面介绍 sort 命令时，每行只有一列数据，实际上多列的情况更常见。下面的文件 colleague.txt 有 3 列，分别是名字、年龄、专业，域分隔符是〈Tab〉键：

```
$ cat colleague.txt
Mike	25	History
Marry	28	English
Jack	23	Math
```

对文件 colleague.txt 排序：

```
$ sort colleague.txt
Jack	23	Math
Marry	28	English
Mike	25	History
```

上面的结果是以整行为单位的升序排序，实际上，几乎就是按照名字的升序排序，因为名字处在每行的最前面。如果希望按照第二列，即按照年龄排序，怎么做？使用选项-k 2 可以对第二列排序，因为第二列是数值，所以加上选项-n 可以确保年龄由小到达排序：

```
$ sort -k 2 -n colleague.txt
Jack        23      Math
Mike        25      History
Marry       28      English
```

按照第三列排序：

```
$ sort -k 3 colleague.txt
Marry       28      English
Mike        25      History
Jack        23      Math
```

可见，是按照第三列的单词的字典顺序排的序。

文件 colleague.txt 的域分隔符是默认的〈Tab〉键。如果不是话，需要用选项-t 来指定域分隔符。

sort 命令的输出显示在屏幕。如果希望 sort 的输出存入文件，可以使用输出重定向，如 sort colleague.txt > colleague_sort.txt。重定向的文件名不能与原文件名重名，即不能运行 sort colleague.txt > colleague.txt，否则文件 colleague.txt 的内容被清空，字节数变为 0。

如果一定让排好序的文件名就叫原来的文件名，可以分两步命令实现，先运行 sort colleague.txt > colleague_sort.txt，再运行 cp colleague_sort.txt colleague.txt。还有一种简便的方法，就是使用 sort 命令的选项-o。-o 用来指定 sort 命令的输出（output）存放的文件名，它可以与原来的名字重名，如：

```
$ sort colleague.txt -o colleague.txt
$ cat colleague.txt
Jack        23      Math
Marry       28      English
Mike        25      History
```

8.7 流编辑器 sed

在字符串替换方面，用 sed 比用 vi（见 2.18 节）更方便。sed 是流编辑器（stream editor）的缩写。sed 的常用格式为：

```
sed [选项] '命令'  文件名
```

8.7.1 替换命令 s

sed 最常用的命令是 s/regexp/replacement/。该命令的作用是，当某部分内容匹配了正则表达式 regexp 时，用 replacement 替换它。前面提到过文件 Linux.txt，把文件 Linux.txt 中的

Linux 换为 UNIX：

```
$ sed 's/Linux/UNIX/' Linux.txt
Hello Jack,
I like UNIX, it's very interesting.
There are many UNIX books.
Best regards,
Mike
```

执行完上面的命令后，文件 Linux.txt 本身不受影响，如果要保留 sed 命令的输出，可以把输出重定向到某个文件。

如果一行有多个 Linux，都能被替换为 UNIX 吗？下面的 echo 命令中含有两个 Linux，通过管道传给 sed 后，发现只有第一个 Linux 被替换为 UNIX：

```
$ echo "I study Linux, I need a Linux book" | sed 's/Linux/UNIX/'
I study UNIX, I need a Linux book
```

实际上替换命令的完整格式是：s/regexp/replacement/flags。在命令的后面加上表示全局（global）的标志（也叫修饰符）g，可以保证一行中的每个匹配的字符串都得到替换：

```
$ echo "I study Linux, I need a Linux book" | sed 's/Linux/UNIX/g'
I study UNIX, I need a UNIX book
```

也可以将 g 换为数字标志，1 就是将第 1 个匹配替换，3 就是将第 3 个匹配替换：

```
$ echo "Linux Linux Linux" | sed 's/Linux/UBUNTU/1'
UBUNTU Linux Linux
$ echo "Linux Linux Linux" | sed 's/Linux/UBUNTU/3'
Linux Linux UBUNTU
```

由于 Linux 系统对字母大小写敏感，下面的命令虽然使用了标志 g，但只有最前面的 Linux 被替换为 UBUNTU：

```
$ echo "Linux LinUX lINux" | sed 's/Linux/UBUNTU/g'
UBUNTU LinUX lINux                    # 最前面的被替换为 UBUNTU
```

标志 i 的作用是忽略字母的大小写，即对大小写不敏感（insensitive）：

```
$ echo "Linux LinUX lINux" | sed 's/Linux/UBUNTU/gi'
UBUNTU UBUNTU UBUNTU            # 不区分字母大小写时，都被替换为 UBUNTU
```

修饰符 g 最常用。还有其他几个修饰符，这里不全部讲解。

下面把文件 Linux.txt 每行的前两个字符替换为一个字符 A，^..表示行首任意两个字符：

```
$ sed 's/^../A/' Linux.txt
Allo Jack,
Alike Linux, it's very interesting.
```

```
Aere are many Linux books.
Ast regards,
Ake
```

把文件 Linux.txt 每行的前两个字符删除，即前两个字符被替换为空：

```
$ sed 's/^..//' Linux.txt          # 注意后两个斜杠紧挨着，表示空
llo Jack,
like Linux, it's very interesting.
ere are many Linux books.
st regards,
ke
```

把文件 Linux.txt 每行的第一个空格直至行尾的字符替换为字符 B：

```
$ sed 's/ .*$/B/' Linux.txt        # 注意斜杠与点“.”之间有一个空格
HelloB
IB
ThereB
BestB
Mike
```

文件 Linux.txt 的最后一行没有空格，所以最后一行的内容还是 Mike，没有变化。文件 Linux.txt 的前 4 行有空格，所以内容发生了变化。正则表达式“ .*$”表示一个空格加任意多个字符直至行尾。以文件 Linux.txt 的第 3 行为例，字符串“ are many Linux books.”“ many Linux books.”“ Linux books.”和“ books.”都满足一个空格加任意多个字符直至行尾的模式要求。为什么正则表达式“ .*$”匹配出“ are many Linux books.”？也就是，为什么最终“ are many Linux books.”被替换为 B，而不是“ many Linux books.”等被替换为 B？答案是，正则匹配是“贪婪的”，在同等条件下，它总是尽可能匹配更多的字符。所以，“ .*$”匹配了最长的“ are many Linux books.”，而不是较短的“ many Linux books.”，更不是最短的“ books.”，以满足正则表达式的贪婪性。

假设有一个很长的脚本，其中有连续的 100 行需要注释掉，如果手工在每行的行首添加井号（#）会比较慢，用 sed 则可以一下子做完。例如，有一个文件 comment_out.txt，为了容易看清，每行标明了行号，内容如下：

```
$ cat comment_out.txt
line 1
line 2
line 3
line 4
line 5
line 6
line 7
line 8
line 9
```

```
line 10
```

要把其中的第 5 到第 7 行注释掉，先指定行号范围，再替换（行首添加#号）：

```
$ sed '5,7s/^/#/' comment_out.txt
line 1
line 2
line 3
line 4
#line 5
#line 6
#line 7
line 8
line 9
line 10
```

使用选项-r 或--regexp-extended 时，sed 支持扩展正则表达式。例如，先将中间为#的一个字符串存入文件 qq，然后先显示#后面的子字符串，再显示#前面的子字符串，中间为一个〈Tab〉键：

```
$ echo 'morning#afternoon' > qq
$ sed -r 's/(.+)#(.+)/\2\t\1/' qq
afternoon       morning
```

(.+)#(.+)为了匹配 morning#afternoon，#就匹配#，.+代表一个或者多个字符，前面的.+匹配 morning，后面的.+匹配 afternoon；\1 和\2 分别代表第 1 个和第 2 个括号里面的匹配，即 morning 和 afternoon，\t 表示〈Tab〉键，则替换结果\2\t\1 显示为 afternoon[Tab 键]morning。如果去掉上面命令中的选项-r，命令将运行失败。

如果希望将显示在屏幕上的 sed 的替换结果保存下来，除了使用输出重定向，还可以使用选项-i，将替换结果直接存入文件。接上面的例子，增加选项-i，将 sed 的运行结果存入文件 qq：

```
$ cat qq
morning#afternoon
$ sed -i -r 's/(.+)#(.+)/\2\t\1/' qq
$ cat qq
afternoon       morning       # 可见文件 qq 的内容变了，原内容消失
```

如果希望保存替换结果的同时，能够备份输入文件，可在-i 后面紧跟备份文件的后缀名，例如：

```
$ echo 'morning#afternoon' > qq
$ sed -i_bak -r 's/(.+)#(.+)/\2\t\1/' qq
$ cat qq
afternoon       morning       # 文件 qq 的内容变了
$ cat qq_bak                  # 备份文件为 qq_bak，保留了原内容
```

```
morning#afternoon
```

8.7.2　其他命令

sed 对输入给它的每一行进行处理，无论内容是否有变化，sed 将默认把每一行显示出来。例如，sed ''（两个单引号紧挨着，单引号之间为空，表示没有 sed 的处理动作，空命令）后面跟文件名，将原封不动地显示该文件，如：

```
$ sed '' Linux.txt
Hello Jack,
I like Linux, it's very interesting.
There are many Linux books.
Best regards,
Mike
```

上面命令的效果与 cat Linux.txt 是一样的。sed 的选项-n 的作用是取消默认输出，下面的命令将没有任何输出：

```
$ sed -n '' Linux.txt
```

sed 中的命令 p 的作用是屏幕打印整行，可以选取行的范围打印，也可以模式匹配打印。下面显示文件 Linux.txt 的前 2 行：

```
$ sed -n '1,2p' Linux.txt
Hello Jack,
I like Linux, it's very interesting.
```

上面命令中的选项-n 是需要的。如果没有选项-n，结果会不一样：

```
$ sed '1,2p' Linux.txt
Hello Jack,
Hello Jack,
I like Linux, it's very interesting.
I like Linux, it's very interesting.
There are many Linux books.
Best regards,
Mike
```

没有选项-n 的话，文件 Linux.txt 的前 2 行显示了两次，一次为 sed 的命令 p 的输出，一次为 sed 的默认输出。最后 3 行为默认输出。

显示文件 Linux.txt 的第 1 行：

```
$ sed -n '1p' Linux.txt
Hello Jack,
```

显示文件 Linux.txt 的第 3 到最后一行（$代表最后一行）：

```
$ sed -n '3,$p' Linux.txt
There are many Linux books.
Best regards,
Mike
```

显示文件 Linux.txt 中包含 Linux 的行，这是模式匹配打印的例子：

```
$ sed -n '/Linux/p' Linux.txt
I like Linux, it's very interesting.
There are many Linux books.
```

sed 中的命令 d 与 p 相反，它的作用是删除整行，可以选取行的删除范围，也可以模式匹配删除。下面删除文件 Linux.txt 的前 2 行：

```
$ sed '1,2d' Linux.txt
There are many Linux books.
Best regards,
Mike
```

删除文件 Linux.txt 中包含 Linux 的行：

```
$ sed '/Linux/d' Linux.txt
Hello Jack,
Best regards,
Mike
```

常用的 sed 编辑命令见表 8-5。

表 8-5　sed 常用编辑命令

命　令	作　用
a\	在当前行之后添加一行或多行，多行时除最后一行外，每行末尾需加续行符\
c\	用新文本替换当前行中的文本，多行时除最后一行外，每行末尾需加续行符\
d	删除文本
i\	在当前行之前插入文本，多行时除最后一行外，每行末尾需加续行符\
l	显示不可打印字符
p	打印文本
r	从文件中读取输入行
s	匹配查找和替换
w	将所选文本写入文件

举一个表 8-5 中的 w 命令例子。将 Linux.txt 中包含 Linux 的行另存为文件 L2.txt：

```
$ cat Linux.txt | sed -n '/Linux/w L2.txt'
```

查看一下 L2.txt：

```
$ cat L2.txt
I like Linux, it's very interesting.
There are many Linux books.
```

再举一个关于 i\命令的例子。在 Linux.txt 中包含 Linux 的行的前面插入内容为 Good Morning 的新行：

```
$ cat Linux.txt | sed '/Linux/i\Good Morning'
Hello Jack,
Good Morning
I like Linux, it's very interesting.
Good Morning
There are many Linux books.
Best regards,
Mike
```

看一下表 8-5 中的命令 l 的解释，就明白 8.3 节中命令 sed -n l 的作用了。表 8-5 还有其他几个命令，这里不一一讲解了。

8.7.3 一行多条命令与保存匹配&

一条 sed 可以含有多个命令操作，使用选项-e，每个-e 后面跟随一个操作。例如，将文件 Linux.txt 中的 Jack 换为 Rose，并且将包含 interest 的行删除：

```
$ sed -e 's/Jack/Rose/' -e '/interest/d' Linux.txt
Hello Rose,
There are many Linux books.
Best regards,
Mike
```

sed 命令的两个单引号之间也可以包含多个操作，中间以分号隔开。例如：

```
$ sed 's/Jack/Rose/;/interest/d' Linux.txt
Hello Rose,
There are many Linux books.
Best regards,
Mike
```

然后，进一步将小写字母全都转换为大写：

```
$ sed 's/Jack/Rose/;/interest/d;s/[a-z]/\u&/g' Linux.txt
HELLO ROSE,
THERE ARE MANY LINUX BOOKS.
BEST REGARDS,
MIKE
```

其中，s/[a-z]/\u&/g 的作用是将小写字母全都转换为大写，就是字符为小写字母（a-z 之

间）时，进行\u&操作，\u 的作用是转为大写字母（uppercase），&的作用是保存已经匹配上的字符（串）以便调用它。看个例子就对&的作用更清楚了。

用 echo 命令显示 Jack：

```
$ echo Jack
Jack
```

下面的命令显然是将 Jack 替换为 Hello：

```
$ echo Jack | sed 's/Jack/Hello/'
Hello
```

下面的命令是在 Jack 前面加上 Hello（而不是用 Hello 替换 Jack），因为&就代表已匹配上的 Jack，最终显示 Hello Jack：

```
$ echo Jack | sed 's/Jack/Hello &/'
Hello Jack
```

用 sed 将大写字母都转为小写时，使用 sed 's/[A-Z]/\l&/g'，其中\l 的作用是转为小写字母（lower case）。

```
$ echo Jack | sed -e 's/Jack/Hello &/' -e 's/[A-Z]/\l&/g'
hello jack
```

8.7.4　sed 的退出状态

grep 命令在找到了它所查找的模式时，退出状态为 0，若没找到，则退出状态为非 0。例如，用 grep 命令列出文件 Linux.txt 中包含 interest 的行：

```
$ grep interest Linux.txt
I like Linux, it's very interesting.
$ echo $?
0                    # 退出状态为 0
```

用 grep 命令查找文件 Linux.txt 中包含 Windows 的行，若找不到，退出状态非 0：

```
$ grep Windows Linux.txt
$ echo $?
1                    # 没找到，退出状态非 0
```

sed 与马上要讲的 awk 都支持模式查找，但无论是否找到，sed 与 awk 的退出状态都为 0，当语法错误时，退出状态为非 0。例如，用 sed 命令显示文件 Linux.txt 中包含 Windows 的行，找不到包含 Windows 的行，退出状态为 0：

```
$ sed -n '/Windows/p' Linux.txt
$ echo $?
0
```

把上面命令中的 p 去掉，遇到语法错误，命令的退出状态为非 0：

```
$ sed -n '/Windows/' Linux.txt
sed: -e expression #1, char 9: missing command
$ echo $?
1
```

8.7.5　sed 脚本

sed 的功能非常强大，前面介绍的内容只是它的冰山一角。通过前面的学习，似乎可以说 sed 是一条交互式的编辑命令，但它也有非交互的一面。sed 本身也是一门脚本语言，它的另一种格式为：

```
sed [选项] -f sed-script  文件名
```

其中 sed-script 就是事先写好的 sed 脚本。在一行当中，可以有多条 sed 命令，但是当命令很多或者很长时，用 sed 脚本更方便一些。脚本 sed_1.sh 中包含了四条命令，第一条的作用是将 Jack 换为 Rose，第二条的作用是将含有 interest 的行删除，第三条的作用是在最后一行（$代表最后一行）的前面加一行 Yours Sincerely，第四条的作用是在最后一行的后面加一行 In Beijing, Friday：

```
$ cat sed_1.sh
s/Jack/Rose/
/interest/d
$i\Yours Sincerely
$a\In Beijing, Friday
```

用命令 sed -f 运行脚本 sed_1.sh 对文件 Linux.txt 进行处理：

```
$ sed -f sed_1.sh Linux.txt
Hello Rose,
There are many Linux books.
Best regards,
Yours Sincerely
Mike
In Beijing, Friday
```

可见，脚本的执行结果与预期相符。将脚本 sed_1.sh 的第一行写为#!/bin/sed -f（与 shell 和 Perl 脚本一样，首行写脚本解释程序），得到脚本 sed_2.sh：

```
$ cat sed_2.sh
#!/bin/sed -f          # 有的系统的路径为/usr/bin/sed，可运行 which sed 查询
s/Jack/Rose/
/interest/d
$i\Yours Sincerely
$a\In Beijing, Friday
```

用 chmod +x 命令增加该 sed 脚本的执行权限，然后，可以像执行一条 Linux 命令那样执行该 sed 脚本：

```
$ sed_2.sh Linux.txt
Hello Rose,
There are many Linux books.
Best regards,
Yours Sincerely
Mike
In Beijing, Friday
```

写 sed 脚本时需要注意：脚本中命令的两边不再使用引号了；用于插入或者替换的文本很长并超过一行时要用续行符号\；每行的末尾不要有多余的空格和〈Tab〉键，特别是续行符的后面不能有。

8.8　文本处理工具 awk

与 sed 一样，awk 并不是 Bash 的一部分，但事实上几乎所有的 Linux 系统都安装了 sed 与 awk，并且，常常见到 Bash 脚本里面有调用 sed 与 awk 的命令。awk 是 Unix/Linux 下一种优秀的用于处理文本的编程语言工具。与其他大多数 Linux 命令不同的是，仅从名字上看，猜不出 awk 的功能。awk 的名称来自它的三个创始人 Alfred Aho、Peter Weinberger 和 Brian Kernighan 的姓氏的首字母。

awk 最基本的命令格式为：

```
awk '命令'  文件名
```

awk 对文件进行逐行扫描并处理。假设有一个文件 team.txt，一共 4 列，内容分别为名字、工号、生日、工资，用 awk 可以显示这个文件的内容：

```
$ awk '{print $0}' team.txt
Jack      3578    1976/07/15    2300
Mike      5346    1972/06/08    3200
Mary      1655    1971/09/26    3600
Mick      1683    1980/11/19    2800
```

其中，$0 代表 awk 读入的文件的每一整行内容，这里的 print 是 awk 中的命令，不是 Linux 命令。命令 awk '{print $0}' team.txt 与命令 cat team.txt 的效果相同，awk 依次读取文件 team.txt 的每一行并且打印出来。

下面的命令只显示名字和工资，并加上序号：

```
$ awk '{print NR, $1 ,$4}' team.txt
1 Jack    2300
2 Mike    3200
3 Mary    3600
```

```
4 Mick      2800
```

其中，$1 表示文件的第 1 列，即名字；$4 表示第 4 列，即工资；NR（Number of the current Record）为 awk 的内置变量，表示当前记录（即当前行）的序号。

下面的例子，awk 依次读取文件 team.txt 的每一行并输出 Good morning，因为 team.txt 一共有四行，所以输出了四次 Good morning：

```
$ awk '{print "Good morning"}' team.txt
Good morning
Good morning
Good morning
Good morning
```

通过上面的例子，可以进一步理解 awk 的“逐行扫描”。多数情况下，awk 是文本输入驱动的。如果要输出 Good morning，运行 awk '{print "Good morning"}'这个没有文本输入的命令是不行的：

```
$ awk '{print "Good morning"}'
                              # 终端屏幕没有输出，也没有系统提示符的出现
```

这个命令将没有任何输出，因为 awk 在等待文本输入（命令中不含有文件名，靠文本输入驱动的 awk 在等待键盘输入），键盘输入任何字符串，就会输出 Good morning：

```
$ awk '{print "Good morning"}'
yes                # 键盘输入
Good morning       # awk 的显示
game               # 键盘输入
Good morning       # awk 的显示
^C                 # Ctrl+C 键中断 awk
```

难道让 awk 做点事情必须要有文本输入驱动吗？其实也不是这样的，后面要讲到的 awk 的 BEGIN 模块可以解决这个问题。

awk 支持正则表达式，基本的命令格式为：

```
awk '查找模式 [命令]' 文件名
```

例如，想知道工号为 3578 的人的信息，用如下命令，其中$2 ~ 3578 表示第二列匹配 3578：

```
$ awk '$2 ~ 3578' team.txt
Jack   3578    1976/07/15   2300
```

想知道 1971 年出生的人的名字，用如下命令：

```
$ awk '/1971/ {print $1}' team.txt
Mary
```

awk 自身有重定向输出功能。下例中，awk 取出了 team.txt 的第 1 列和第 4 列，另存为

wage.txt 文件：

```
$ awk '{print $1, $4 > "wage.txt"}' team.txt
$ cat wage.txt
Jack 2300
Mike 3200
Mary 3600
Mick 2800
```

前面提到了两种 awk 的基本命令格式，还有一种较常用的格式为：

```
awk [-F 域分隔符] '命令' 文件名
```

-F 后面带域分隔符，默认的域分隔符是空格。前面的几个例子中，数据的域分隔符是空格，所以 awk 命令都省略了选项-F。

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
```

显然，文件/etc/passwd 的域分隔符是:，要查看 Linux 账号和对应的默认 shell，用下面的命令：

```
$ awk -F: '{print $1, $7}' /etc/passwd
root /bin/bash
daemon /bin/sh
bin /bin/sh
sys /bin/sh
sync /bin/sync
```

实际上-F 后面跟的域分隔符指的是输入域分隔符（/etc/passwd 是 awk 命令的输入文件），输入域分隔符也可以由 awk 的内置变量 FS 指定；相应地，输出域分隔符由 awk 的内置变量 OFS 指定，它们的默认值都是空格。例如，查看 Linux 账号和对应的默认 shell，输出的两列用#分隔：

```
$ awk '{FS=":"; OFS="#"; print $1, $7}' /etc/passwd
root#/bin/bash
daemon#/bin/sh
bin#/bin/sh
sys#/bin/sh
sync#/bin/sync
```

与 Perl 语言一样，awk 有 BEGIN 和 END 模块。简单地说，BEGIN 模块总是最先执

行，END 模块总是最后执行。例如，可以使用 BEGIN 模块给 team.txt 加文件头：

```
$ awk 'BEGIN{ print "Name    Number birthday       wage"} {print $0}' team.txt
Name        Number birthday      wage
Jack        3578     1976/07/15   2300
Mike        5346     1972/06/08   3200
Mary        1655     1971/09/26   3600
Mick        1683     1980/11/19   2800
```

下面使用 END 模块计算并打印平均工资，由结果可知，这几个人的平均工资为 2975 元：

```
$ awk '{sum=sum+$4} END{print sum/NR}' team.txt
2975
```

不管位置的先后，BEGIN 模块最先执行，END 模块最后执行。如下例所示，awk 先执行 BEGIN 模块，打印 Morning，然后读文件 team.txt，该文件一共四行，所以打印了四次 Mary，最后执行 END 模块，打印 Bye：

```
$ awk '{print "Mary"} END{print "Bye"} BEGIN{ print "Morning"}' team.txt
Morning
Mary
Mary
Mary
Mary
Bye
```

前面讲过，在多数情况下，awk 需要文本输入来驱动。在没有文本输入的情况下要想输出 Good morning，可以用 BEGIN 模块：

```
$ awk 'BEGIN{print "Good morning"}'
Good morning
```

awk 具有与 C 语言十分类似的流程控制功能，如 if 和 if/else 条件控制，while、do-while 和 for 循环，用于跳转或者退出的命令 break、continue、next 和 exit 等。例如，打印工资高于 3000 元的人的信息：

```
$ awk '{if($4>3000){print $0}}' team.txt
Mike        5346     1972/06/08   3200
Mary        1655     1971/09/26   3600
```

可知，一共有两个人的工资高于 3000 元。下面的例子，打印出第一个工资高于 3000 元的人的信息后，马上执行 exit，程序退出了，所以只打印了一个人的信息：

```
$ awk '{while($4>3000){print $0;exit}}' team.txt
Mike        5346     1972/06/08   3200
```

awk 有内置的数学函数，这些函数是 Bash 所没有的，见表 8-6。

表 8-6　awk 的数学函数

名　称	返　回　值	名　称	返　回　值
atan2(y,x)	y/x 的反正切弧度值	rand()	0 到 1 之间的随机数
cos(x)	以 x 为弧度的余弦	sin(x)	以 x 为弧度的正弦
exp(x)	e 的 x 次幂	sqrt(x)	x 的平方根
int(x)	取 x 的整数部分	srand(x)	x 是 rand()函数的种子
log(x)	以 e 为底的 x 的对数		

例如，int(15.68)输出整数部分 15，sqrt(3)输出 3 的平方根 1.73205：

```
$ awk 'BEGIN{print int(15.68), sqrt(3)}'
15    1.73205
```

因为 team.txt 有四行，所以下面的命令打印出四个 0 到 1 之间的随机数：

```
$ awk '{print rand()}' team.txt
0.411026
0.120443
0.280994
0.663271
```

awk 提供了字符串处理函数，例如，index("Mississippi", "is")返回子字符串 is 在字符串 Mississippi 中首次出现的位置；substr("707-555-1111", 1, 3)返回字符串 707-555-1111 从第 1 个到第 3 个字符组成的子字符串；length("abcdefghij")返回字符串 abcdefghij 的长度。具体执行结果如下：

```
$ awk 'BEGIN { print index("Mississippi", "is")}'
2
$ awk 'BEGIN { print substr("707-555-1111", 1, 3)}'
707
$ awk 'BEGIN { print length("abcdefghij")}'
10
```

awk 还有其他字符串处理函数，这里就不一一列举了。

与 Perl 语言一样，awk 提供了系统调用函数 system。例如，awk 调用 Linux 命令 date，查看日期和时间：

```
$ awk 'BEGIN{system("date")}'
Thu Jun 13 00:54:38 EDT 2013
```

awk 还可以应用 system 函数调用其他 shell 或者 Perl 脚本，这为混合编程提供了方便。

前面的例子，给人一种感觉，似乎 awk 只有命令行形式。其实，awk 也可以像 Bash 一样写成脚本，格式为：

```
awk [选项] -f awk-script [文件名]
```

下面脚本 factorial_1.awk 的功能是计算并打印出 6 到 10 的阶乘：

```
$ cat factorial_1.awk
function factorial(n)         # 计算阶乘的递归函数，awk 支持递归
{
    if(n<0)
    {
        print("n must >=0");
    }
    else if (n <=1)
    {
        return 1;
    }
    else
    {
        return n*factorial(n-1);
    }
}
BEGIN{
    for(i=6;i<=10;i++)
    {
        printf("%d!=%d\n",i,factorial(i));
    }
}
```

脚本 factorial_1.awk 中使用了格式化打印命令 printf，awk 的 printf 与 C 语言的 printf 在基本语法上几乎完全一致。

执行 awk -f 命令，可打印出 6 到 10 的阶乘：

```
$ awk -f factorial_1.awk
6!=720
7!=5040
8!=40320
9!=362880
10!=3628800
```

如果将 awk -f 加入到脚本的第一行，得到脚本 factorial_2.awk：

```
$ cat factorial_2.awk
#!/usr/bin/awk -f             # 有的系统 awk 的全路径为/bin/awk，可运行 which awk 查询
function factorial(n)
{
    ……
}
BEGIN{
    for(i=6;i<=10;i++)
    ……
}
```

那么，直接运行 factorial_2.awk 脚本，即可打印出 6 到 10 的阶乘：

```
$ factorial_2.awk
6!=720
7!=5040
8!=40320
9!=362880
10!=3628800
```

awk 有两个与 C 语言类似的内置变量 ARGC 和 ARGV，用于读取命令参数。awk 还有内置数组变量 ENVIRON，可用来读取当前系统的环境信息等。awk 是一门脚本编程语言，内容很多，这里不展开讲解。

8.9　格式化打印命令 printf

内置命令 printf 用来进行格式化的打印输出，语法格式如下：

```
printf [-v var] format [参数]
```

格式化字符串 format 一般用双引号括起来，它可以包含三种类型的字符。第一种为普通字符；第二种为转义字符，由反斜杠引出，最常见的、前面也提到过的为\n，用来换行；第三种为格式指示符（format-specifier），由百分号引出，最常见的为用来打印整数的%d 和打印字符串的%s。

例如，显示问候语，可以运行 echo "Good morning"，用 printf 命令的话，运行：

```
$ printf "Good morning\n"
Good morning
```

echo 命令默认带着换行符，而 printf 命令默认不带，所以，在每条 printf 命令中加换行符\n 几乎是必须的。如果不加，那么命令的输出与系统提示符将出现在同一行：

```
$ printf "Good morning"
Good morning$                    # Good morning 与提示符$处在同一行，因为没有\n
```

下面是关于格式指示符的例子，显示年龄和名字：

```
$ printf "My age is %d \n" 20
My age is 20
$ printf "My name is %s \n" Mike
My name is Mike
```

也许读者会问，直接运行命令 echo "My age is 20"不就可以吗，为什么这么复杂？printf 确实比 echo 复杂，但功能强大，echo 不可能替代 printf。

printf 的输出默认显示在屏幕上，用选项-v 可以让 printf 的输出不显示在屏幕上，而是记录在一个变量里。例如：

```
$ printf -v greeting "Good morning\n"
```

这时，屏幕没有显示输出，Good morning 被记录在变量 greeting 里。显示 greeting 的值就知道了：

```
$ echo $greeting
Good morning
```

关于转义字符，在第 2 章讲 echo 的时候已经讲过，printf 命令使用的转义字符与 echo 命令的基本一致，这里不再重复。关于格式指示符，见表 8-7。

表 8-7 printf 格式指示符

格式指示符	描　述
%b	相应的参数被视为含有要被处理的转义字符的字符串
%c	ASCII 字符
%s	字符串
%d	有符号十进制整数
%i	有符号十进制整数，同%d
%u	无符号十进制整数
%o	无符号八进制整数
%x	无符号十六进制整数，使用 0-9，a-f 表示十六进制数
%X	无符号十六进制整数，使用 0-9，A-F 表示十六进制数
%f	浮点数，格式为 [-]w.precision，w 为显示的总宽度，precision 表示精度
%e	浮点数，格式为 [-]w.precision **e**[+/-]dd
%E	浮点数，格式为 [-]w.precison **E**[+/-]dd
%g	%e 或者%f 转换，看哪一个较短则删除结尾的零，默认为 6 位有效位
%G	%E 或者%f 转换，看哪一个较短则删除结尾的零，默认为 6 位有效位
%%	百分号本身

例如，十进制的 28，对应的八进制数值为 34、十六进制数值为 1c（小写字母）或 1C（大写字母）：

```
$ printf "%d %i %o %x %X\n" 28 28 28 28 28
28 28 34 1c 1C
```

同 C 语言一样，数字 0 开头的数为八进制数，0x 或者 0X 开头的数为十六进制数，例如：

```
$ printf "%d\n" 030
24                    # 八进制 30 对应十进制 24
$ printf "%d\n" 0x3F
63                    # 十六进制 3F 对应十进制 63
```

下面对比一下%s 与%b，就会明白%b 的作用。

```
$ printf "%s\n" "Good\tmorning"
Good\tmorning                    # 字符串原样打印
$ printf "%s\n" "Good\nmorning"
```

```
Good\nmorning                 # 字符串原样打印
$ printf "%b\n" "Good\tmorning"
Good        morning           # 字符串中的\t 为转义字符，表示〈Tab〉键
$ printf "%b\n" "Good\nmorning"
Good
morning                       # \n 为转义字符，表示换行，Good 和 morning 不在同一行显示
```

%f 用来显示浮点数，默认精度为保留到小数点后 6 位：

```
$ printf "%f\n" 2.3567847
2.356785
```

使用%10.4f 时，10 表示显示数值的总宽度，4 表示小数点后的保留位数，得到 2.3568，前面有四个空格，即，向右对齐，总宽度为 10：

```
$ printf "%10.4f\n" 2.3567847
    2.3568
```

使用%-10.4f 时，是向左对齐，总宽度仍然为 10，2.3568 的后面有四个空格。为了能明显看出 2.3568 的后面有四个空格，在\n 的前面加一个分号：

```
$ printf "%-10.4f;\n" 2.3567847
2.3568    ;
```

使用%15.4e，向右对齐，23.567847 显示为 2.3568 乘以 10 的一次方，即，2.3568e+01：

```
$ printf "%15.4e;\n" 23.567847
     2.3568e+01;
```

使用%-15.4E，总宽度为 15，向左对齐，0.00023567847 显示为 2.3568 乘以 10 的负 4 次方，即，2.3568E-04：

```
$ printf "%-15.4E;\n" 0.00023567847
2.3568E-04     ;
```

使用%10.4f，小数点后保留 4 位，不够 4 位则补 0：

```
$ printf "%10.4f\n" 2.3
    2.3000
$ printf "%10.4e\n" 2.3
2.3000e+00               # 这里 2.3 显示为 2.3000 乘以 10 的 0 次方
```

使用%-10.4G，后面的 0 不显示，总宽度为 10：

```
printf "%-10.4G;\n" 2.3
2.3       ;
```

以上只介绍了 printf 的一小部分功能。内置命令 printf 与 C 语言的函数 printf 的功能类似，学习过 C 语言的人掌握它非常容易。

第 9 章　进程与作业

日常工作中的一些脚本或命令不是很快就能执行完毕，而是需要一段时间，有些脚本在后台长期运行。每个脚本或命令一旦运行，都有相应的进程号。Linux 是一个多任务操作系统，可以有多个进程同时运行，正在运行的一个或多个相关的进程称为一个作业。下面介绍进程管理及作业控制的相关命令。

9.1　查看进程命令 ps

命令 ps 用来查看正在系统中运行的进程的信息。不带选项的 ps 命令查看当前用户的进程信息，例如：

```
$ ps
  PID TTY          TIME    CMD
 9165 pts/73       00:00:00 bash
 9279 pts/73       00:00:00 ps
24735 pts/73       00:00:00 1355294669.2545
24737 pts/73       00:00:00 interactive_job
24795 pts/73       00:00:00 perl
24968 pts/73       00:00:00 tcsh
```

一共有 4 列信息，分别是：进程标识号 PID(Process ID)，终端号 TTY，进程占用的 CPU 时间，相应的命令。上例中输出的第二行“9165 …… bash”就是当前的 shell 命令窗口的进程，第三行“9279 …… ps”就是刚刚运行的 ps 命令。

加上选项-f 的话，可以显示更多的信息（f 是 full-format listing 的意思），包括父进程标识号 PPID(Parent PID)，进程启动时间 STIME 和命令参数：

```
$ ps -f
UID       PID     PPID  C   STIME    TTY      TIME CMD
maggie   9165    24968  0   12:51    pts/73   00:00:00 bash
maggie   9302     9165  0   12:52    pts/73   00:00:00 ps -f
maggie  24735    24629  0   07:44    pts/73   00:00:00 /bin/sh /home/maggie/
1355294669.2545
maggie  24737    24735  0   07:44    pts/73   00:00:00 /bin/sh /opt/exec/interactive_job
maggie  24795    24737  0   07:44    pts/73   00:00:00 /usr/bin/perl -w u_tool
maggie  24968    24795  0   07:44    pts/73   00:00:00 -csh
```

如果要查看所有账户的进程，用命令 ps -ef；如果要查看某个账户的进程，用命令 ps -ef | grep <账户名>。

9.2　挂起进程〈Ctrl+Z〉键

运行 vi test.txt 编辑文件，假设编辑到一半的时候，想运行某条命令，有如下几种办法：

1）将 test.txt 保存，然后退出 vi，看到命令行提示符后，运行命令；

2）重新打开一个命令行窗口，运行命令；

3）在 vi 的底行模式下输入：!命令，例如，输入!date 后按〈回车〉键即可查看当前日期和时间；

还有一种方法，是将 vi 挂起。在用 vi 编辑文件 test.txt 时，按〈Esc〉键，确保 vi 处在命令模式下，再按〈Ctrl+Z〉键，可以将 vi 挂起：

```
$ vi test.txt
[1]+  Stopped                 vi test.txt
$                          # 按 Ctrl+Z 键，挂起了 vi 并得到了提示符
```

按〈Ctrl+Z〉键，挂起了 vi 并得到了命令行提示符，这时就可以执行其他命令了。运行内置命令 jobs，可以查看作业状态：

```
$ jobs
[1]+  Stopped                 vi test.txt
```

1 是作业号，Stopped 是作业的一种状态，表示暂停（注意，不是完全停止）。加上选项-l 可显示作业的进程 ID：

```
$ jobs -l
[1]+  3038 Stopped                 vi test.txt
```

用选项-p 时，仅显示进程 ID：

```
$ jobs -p
3038
```

用选项-s 时，仅显示状态为 Stopped 的作业：

```
$ jobs -s
[1]+  Stopped                 vi test.txt
$ jobs -sl
[1]+  3038 Stopped            vi test.txt
```

作业号只是在当前的窗口有效，从 1 开始编号，进程 ID 在整个系统中有效。重新打开一个命令窗口，执行 jobs，没有输出，而查找 ID 为 3038 的进程，是可以查到的：

```
$ jobs
$ ps -ef | grep 3038
user       3038  2986  0 Jan16 pts/1    00:00:00 vi test.txt
user       3101  3040  0 00:14 pts/2    00:00:00 grep --color=auto 3038
```

回到刚才的命令窗口，运行不带选项和参数的命令 cat，系统等待用户输入。不输入任何字符，按〈Ctrl+Z〉键，将 cat 挂起：

```
$ cat
^Z
[2]+   Stopped                    cat
```

现在有两个作业：

```
$ jobs -l
[1]-        3038 Stopped                    vi test.txt
[2]+        3108 Stopped                    cat
```

9.3　前台 fg 与后台 bg

在 3.11.2 节讲过，命令的后面带&即可将命令直接放在后台执行。例如，find 命令有时用时较长，特别是从根目录开始查找时，这时可以将其放在后台执行。下面的后台命令从根目录开始查找后缀名为 doc 的文件：

```
$ find / -name "*.doc" 2>/dev/null &
[3] 3578
```

则当前命令窗口有 3 条后台命令，find 命令的状态为运行中（Running），用选项-r 时，命令 jobs 仅显示状态为 Running 的作业：

```
$ jobs
[1]         Stopped                 vi test.txt
[2]-        Stopped                 cat
[3]+        Running                  find / -name "*.doc" 2> /dev/null &
$ jobs -r
[3]+        Running                  find / -name "*.doc" 2> /dev/null &
```

仔细查看 jobs 命令的输出，可以看到+和−。+代表最近（最新）的作业，也称为当前作业，−代表前一个作业。

现在当前窗口有 3 条后台命令，如果希望把某条命令拿到前台（foreground），可用内置的 fg 命令，格式为：

```
fg [作业号]
```

例如，想把 vi 命令拿到前台，继续编辑 test.txt 文件，执行：

```
$ fg %1
```

如果把最近的被放在后台的作业（即当前作业）放到前台，fg 命令参数为%+或者%%。如果把第二近后台作业放到前台，参数为%−。如果事先知道命令的名字，例如想把 vi 命令拿到前台，并且当前窗口的后台命令当中只有一个 vi 命令的话，fg 命令参数可以为%vi。引用后台作业的方法见表 9−1。

表 9-1　引用后台作业的方法

引　　用	所指的后台作业	引　　用	所指的后台作业
%N	编号为 N 的作业	%+	最近的被放在后台的作业
%string	命令以 string 开头的作业	%%	同上
%?string	命令包含 string 的作业	%-	第二近的被放在后台的作业

命令 bg 的作用是把作业放到后台（background）。重新打开一个 Terminal，从根目录下开始查找名字以.gz 结尾的文件，立刻按〈Ctrl+Z〉键挂起它：

```
$ find / -name "*.gz" 2>/dev/null
^Z
[1]+        Stopped                  find / -name "*.gz" 2> /dev/null
```

运行 jobs，可见当前有一个作业，状态为 Stopped：

```
$ jobs
[1]+        Stopped                  find / -name "*.gz" 2> /dev/null
```

用命令 bg，把该作业放到后台执行：

```
$ bg %1
[1]+ find / -name "*.gz" 2> /dev/null &
```

运行 jobs，可见该作业的状态变为 Running：

```
$ jobs
[1]+        Running                  find / -name "*.gz" 2> /dev/null &
```

如果 fg（bg）命令不带任何参数，那么当前作业将被放到前（后）台，即无参数的命令 fg（bg）与命令 fg %+（bg %+）效果是一样的。

把作业放到前台或者后台执行有更简洁的方式。假设作业号为 1，将其放到前台可以执行%1，与命令 fg %1 效果相同；放到后台的话可以执行%1 &，与命令 bg %1 效果相同。

9.4　发送信号命令 kill

从字面上看，内置命令 kill 用来“杀掉”某种东西，它确实有这方面的功能。但 kill 主要用于给作业或者进程发送信号。kill 加选项-l，可以列出信号名称与编号：

```
$ kill -l
 1) SIGHUP        2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT       7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV      12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT    17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN      22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM    27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
```

```
31) SIGSYS……
……
63) SIGRTMAX-1 64) SIGRTMAX
```

选项-l 的后面跟信号的编号时，kill 命令输出相应的信号名：

```
$ kill -l 13
PIPE
$ kill -l 21
TTIN
```

选项-l 的后面跟信号名时，kill 命令输出相应的信号编号：

```
$ kill -l SIGABRT
6
$ kill -l PIPE
13
```

选项-l 的后面跟不正确的信号名或者信号编号时，系统会提示出错：

```
$ kill -l PIXYZ
bash: kill: PIXYZ: invalid signal specification
$ kill -l 89
bash: kill: 89: invalid signal specification
```

用命令 kill 给作业或者进程传递信号的格式为：

```
kill [-s 信号名 | -n 信号编号 | -信号名 | -信号编号] 作业号或进程 ID
```

如果 kill 命令当中没有指定信号时，默认发送的是信号 SIGTERM。

从根目录下开始查找名字以.gz 结尾的文件，马上按〈Ctrl+Z〉键挂起它：

```
$ find / -name "*.gz" 2>/dev/null
^Z
[1]+  Stopped                 find / -name "*.gz" 2> /dev/null
```

运行 jobs，可见当前有一个作业，状态为 Stopped：

```
$ jobs
[1]+  Stopped                 find / -name "*.gz" 2> /dev/null
```

如果想终止该作业，运行 kill %1：

```
$ kill %1
```

再运行 jobs，可见作业的状态变为 Terminated（因为收到了默认信号 TERM，见表 9–2）：

```
$ jobs
[1]+  Terminated              find / -name "*.gz" 2> /dev/null
```

由于各种原因，用带默认信号 TERM 的 kill 命令有时会终止不了某个进程。信号 KILL，即编号为 9 的信号可用来强行终止进程或者作业。例如，运行 vi 命令，直接将其放到后台：

```
$ vi email.txt &
[1] 2945
```

用信号 KILL 强行终止它，运行命令：

```
$ kill -s SIGKILL 2945
```

把进程 ID 2945 换为作业号%1，效果是一样的。上面的命令还可以改为 kill -s KILL %1，kill -KILL %1，kill -n 9 %1，或者 kill -9 %1，效果都是一样的，实际工作中 kill -9 比较常用。这样理解容易记住它：9 在单个数字里面是“老大”，它可以“搞定”一切。这样理解只是为了便于记忆，在 9.6 节末会讲解用信号 9 可以强行终止进程的原因。

常用的信号见表 9−2。

表 9−2　常用信号

信　　号	作　　用
信号 1，HUP	挂起——关闭进程通信连接
信号 2，INT	中断——通知进程退出（〈Ctrl+C〉键）
信号 3，QUIT	退出——强制进程退出（〈Ctrl+\〉键）
信号 6，ABRT	放弃进程
信号 9，KILL	终止进程，该信号不能被截获
信号 15，TERM	软件终止，kill 默认信号，通知进程中运行的程序退出
信号 20，TSTP	挂起正在运行的进程（〈Ctrl+Z〉键）

9.5　等待命令 wait

内置命令 wait 的格式为：

```
wait [id]
```

id 是指作业号或者进程 ID。当 wait 后跟一个不存在的作业号或者进程 ID 时，会有错误提示：

```
$ wait %8
bash: wait: %8: no such job
$ wait 2756
bash: wait: pid 2756 is not a child of this shell
```

wait 不带参数时，它的作用是等待当前 shell 的子进程都结束。使用 wait 可以实现并发多进程操作，先看一个没有 wait 的脚本：

```
$ cat wait_1.sh
```

```
#!/bin/bash
rm -fr qq                    # 如果文件 qq 已经存在，先删除它
for (( i=0; i<5; i++ )); do
{
    sleep 2
    echo $i >> qq
    echo "done!"
}
done
```

脚本中有一个 for 循环，一共 5 轮，每循环一次，脚本暂停 2s（sleep 2），然后将 i 的值存入文件 qq 并显示 done。这个脚本的运行时间应该是 10s，下面计时运行它：

```
$ time wait_1.sh
done!
done!
done!
done!
done!

real   0m10.019s
user   0m0.004s
sys    0m0.008s
```

可见，脚本 wait_1.sh 实际用时为 10.019s。文件 qq 的内容比较容易理解，行的内容依次为 0 到 4。

```
$ cat qq
0
1
2
3
4
```

从 3.11.8 节可知，大括号里面的命令组是在当前 shell 里面运行。如果将脚本 wait_1.sh 中大括号的后面加后台运行符号&，那么 5 组（因为 i 从 0 循环到 4）大括号里面的命令将在 5 个子 shell 里面运行（几乎同时被放到后台运行）。for 循环的后面加命令 wait，保证脚本的子进程都结束后，脚本本身才结束运行。稍加修改，得到脚本 wait_2.sh：

```
$ cat wait_2.sh
#!/bin/bash
rm -fr qq
for (( i=0; i<5; i++ )); do
{
    sleep 2; echo $i >> qq && echo "done!"
} &
```

```
done
wait
```

脚本 wait_2.sh 的运行时间应该是 2s：

```
$ time wait_2.sh
done!
done!
done!
done!
done!

real  0m2.030s
user  0m0.008s
sys   0m0.000s
```

实际用时为 2.030s。文件 qq 的内容不一定为 0 到 4，因为 5 组命令几乎同时运行，并不一定是 i=0 对应的最先运行，i=4 对应的最后运行。查看一下文件 qq 的内容：

```
$ cat qq
3
4
2
1
0
```

再运行一次 wait_2.sh，文件 qq 的内容可能会又不一样：

```
$ wait_2.sh
done!
done!
done!
done!
done!
$ cat qq
2
3
0
1
4
```

9.6　捕获信号命令 trap

多数用户习惯使用〈Ctrl+C〉键来中断正在运行的程序。如果在按下〈Ctrl+C〉键时，希望程序不仅停止运行，还能做其他动作，如给用户一些提示、清理程序运行中产生的垃圾

文件等；或者希望程序能忽略〈Ctrl+C〉键。这样的要求可以通过内置命令 trap 实现，它的常用格式为：

```
trap commands signals
```

commands 代表一条或多条命令，用分号间隔命令；signals 代表一个或多个信号，信号之间要加空格。trap 命令的作用是为 signals 设置陷阱，当 shell 收到 signals 中的某个信号时，就运行 commands。

脚本 non_stop_while_1.sh 中包含一个无穷 while 循环，运行它，屏幕上将一直显示 I study bash，直到按下〈Ctrl+C〉键才停止。

```
$ cat non_stop_while_1.sh
#!/bin/bash
while ( true )
do
    echo "I study bash"
done
```

在脚本中加入一句 trap 命令，得到 non_stop_while_2.sh：

```
$ cat non_stop_while_2.sh
#!/bin/bash
trap 'echo I stop now. Bye-bye.; exit' 2
while ( true )
do
    echo "I study bash"
done
```

trap 命令的作用是，当收到信号 2（即 SIGINT，也就是用户按下〈Ctrl+C〉键）时，显示 I stop now. Bye-bye.，然后执行命令 exit，退出脚本的运行。试一试该脚本：

```
$ non_stop_while_2.sh
I study bash
……
I study bash
I study bash
^CI stop now, Bye-bye.            # 用户按下〈Ctrl+C〉键终止脚本运行
```

使用 trap 命令，还能忽略信号。trap '' 2 的作用是，当收到信号 2 时，不做任何事情（因为两个单引号之间为空命令），就是忽略信号 2。把上面的脚本中的 trap 命令改为 trap '' 2 之后，得到下面的脚本：

```
$ cat non_stop_while_3.sh
#!/bin/bash
trap '' 2
while ( true ); do
    echo "I study bash"
```

```
done
```

运行脚本 non_stop_while_3.sh，再按〈Ctrl+C〉键，无穷循环不会终止，因为〈Ctrl+C〉键被忽略了。如何终止它呢？使用本章前面刚刚讲过的知识，按下〈Ctrl+Z〉键将它挂起，再运行命令 kill %%或者 kill %+终止刚刚挂起的作业。

命令 trap -l 的结果与 kill -l 相同：

```
$ trap -l
 1) SIGHUP       2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
 6) SIGABRT      7) SIGBUS      8) SIGFPE      9) SIGKILL    10) SIGUSR1
11) SIGSEGV     12) SIGUSR2    13) SIGPIPE    14) SIGALRM    15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD    18) SIGCONT    19) SIGSTOP    20) SIGTSTP
21) SIGTTIN     22) SIGTTOU    23) SIGURG     24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF    28) SIGWINCH   29) SIGIO      30) SIGPWR
31) SIGSYS……
……
63) SIGRTMAX-1          64) SIGRTMAX
```

命令 trap -p 或者不带选项和参数的 trap 的作用是列出已经设置的与每个信号相关的 trap 命令，例如：

```
$ trap 'cmd1;cmd2' 6 15          # 设置陷阱，可把 6 和 15 换为信号名 ABRT 和 TERM
$ trap -p                        # 列出已设置的陷阱。去掉-p，运行 trap，效果相同
trap -- 'cmd1;cmd2' SIGABRT
trap -- 'cmd1;cmd2' SIGTERM
```

对某信号的处理，想恢复默认设置的话，运行命令 trap 加该信号。如：

```
$ trap 6                         # 对 SIGABRT 恢复默认设置
$ trap -p
trap -- 'cmd1;cmd2' SIGTERM
```

注意 trap 6 和 trap '' 6 是不一样的，看下面的例子：

```
$ trap '' 6
$ trap -p
trap -- '' SIGABRT
trap -- 'cmd1;cmd2' SIGTERM
```

运行 trap '' 6 之后，shell 收到信号 6 时会忽略它（两个单引号之间为空）。运行 trap 6 之后，shell 收到信号 6，会按照系统默认的方式处理该信号，原先系统默认怎么处理就怎么处理。

当在函数里定义 trap 的时候需要注意，在函数被调用之前，trap 是不生效的。

信号 SIGKILL 和 SIGSTOP 不能被捕获、阻塞或忽略，就是说，对这两个信号设置陷阱是无效的。命令 kill -19（或者-STOP，或者-SIGSTOP）加进程号或作业号可强行暂停进程或作业。9.4 节讲过，终止不掉的进程用 kill -9（或者-KILL，或者-SIGKILL）加进程号即可

强行终止，是因为信号 KILL 不能被捕获，并不是因为 9 在单个数字中是“老大”。

9.7 移除作业的命令 disown

命令 disown 的格式为：

```
disown [-h] [-ar] [作业号]
```

命令 disown 加作业号不是用来停止某个作业（那是 kill 命令的任务），而是从当前 shell 的作业表中移除该作业，移除后，用 jobs 命令查不到它了，但是用 ps 命令仍然可以查到它的相应进程。

下面的脚本包含一个无穷循环，不断地将最新时间写入文件 latest_time.txt：

```
$ cat while_latest_time.sh
#!/bin/bash
date > latest_time.txt
while true
do
    sleep 0.5
    date >> latest_time.txt
done
```

将该脚本放在后台执行：

```
$ while_latest_time.sh &
[1] 1953
$ jobs
[1]+   Running                    while_latest_time.sh &
```

每隔几秒执行 ls -l latest_time.txt 查看它的字节数，或者执行 tail -f latest_time.txt（按〈Ctrl+C〉键可停止 tail -f 命令的显示），可以看到该文件的内容在不断增加：

```
$ tail -f latest_time.txt
Mon Jul 29 03:32:16 EDT 2013
Mon Jul 29 03:32:17 EDT 2013
Mon Jul 29 03:32:17 EDT 2013
Mon Jul 29 03:32:18 EDT 2013
Mon Jul 29 03:32:18 EDT 2013
Mon Jul 29 03:32:19 EDT 2013
^C
```

运行 disown 加作业号可从作业表中移除该作业（不指定作业号时，表示移除当前作业）。移除后，用 jobs 命令查不到它了：

```
$ disown %1
$ jobs
$                 # jobs 命令结果为空
```

用 ps 命令可以查到它：

```
$ ps -ef | grep while_latest_time | grep -v 'grep'
user         1953    1729    0 03:30 pts/0        00:00:01 /bin/bash ./while_latest_time.sh
```

可见，它还在运行，进程 ID 为 1953。执行命令 tail -f latest_time.txt 可以明显知道脚本还在运行。需要尽快终止 1953，否则，文件 latest_time.txt 会越来越大，把硬盘空间占满。

```
$ kill 1953
```

这时，再执行 tail -f latest_time.txt，可以看到文件内容不再变化，说明进程确实停止了。

在不指定作业号时，disown 命令的选项-a 的作用是移除当前 shell 里的所有（all）作业。disown 命令的选项-r 的作用是移除当前 shell 里处在 Running 状态的作业。

disown 命令的选项-h 的作用是使作业忽略 SIGHUP 信号。就是说，如果希望某作业在 shell 收到 SIGHUP 信号时仍能继续运行，可以使用命令：disown -h <作业号>。

命令 disown -ah 的作业是使所有的作业忽略 SIGHUP 信号；命令 disown -rh 的作业是使状态为 Running 的作业忽略 SIGHUP 信号。

9.8　暂停 shell 的命令 suspend

在一个 shell 窗口里，用〈Ctrl+Z〉键可将正在前台运行的程序挂起。如果要将 shell Terminal 本身挂起，按〈Ctrl+Z〉键不起作用，有另外的命令做这件事。

用命令 suspend 可以暂停当前 shell Terminal，直到它收到一个 SIGCONT 信号。这条命令默认对登录 shell 无效。加上选项-f，则可以暂停登录 shell 的运行。

下面挂起一个 shell，再恢复它。内置特殊变量$$（见表 4-2）表示当前 shell 进程 ID。查看一下，可知当前 shell 的进程 ID 是 2620：

```
$ ps | grep $$
 2620 pts/4        00:00:00 bash
```

进入登录 shell，进程 ID 是 2979：

```
$ bash --login
$ ps | grep $$
 2979 pts/4        00:00:00 bash
```

进程 2979 是 2620 的子进程：

```
$ ps -ef | grep $$ | grep login | grep bash
user         2979    2620    0 12:15 pts/4        00:00:00 bash --login
```

用命令 suspend 无法挂起登录 shell，用命令 suspend -f 挂起登录 shell：

```
$ suspend
```

```
bash: suspend: cannot suspend a login shell
$ suspend -f
[1]+  Stopped                 bash --login
```

查看当前 shell 进程 ID，可知登录 shell（进程 ID 是 2979）被挂起，回到了原来的 shell（进程 ID 是 2620）：

```
$ ps | grep $$
 2620 pts/4    00:00:00 bash
```

用 jobs 命令查看，可知登录 shell 被挂起，状态为 Stopped：

```
$ jobs -l
[1]+  2979 Stopped (signal)       bash --login
```

将被挂起的登录 shell 放到前台运行，则又返回到刚才的登录 shell（进程 ID 是 2979）：

```
$ fg %1
bash --login
$ ps | grep $$
 2979 pts/4    00:00:00 bash
```

第 10 章　高 级 话 题

通过前面几章的学习，读者可以写出简单的脚本并看懂别人的脚本。本章主要集中了 Bash 中相对高级的一些应用。

10.1　二次扫描命令 eval

内置命令 eval 的格式为：

```
eval [参数]
```

第一次扫描，eval 会扫描参数，如果有变量，就进行变量值的替换。第二次将参数作为 shell 命令，执行它。eval 后面常常跟一条命令，所以其命令格式也常认为是：

```
eval [命令]
```

例如：

```
$ eval echo "I study bash"
I study bash
```

从上面的例子还看不出 eval 有什么作用。假设有个包含一句话的文件 a.txt：

```
$ cat a.txt
I love bash.
$ MY_FILE="cat a.txt"
$ echo $MY_FILE
cat a.txt
```

上面的命令的输出是 cat a.txt，还看不见文件 a.txt 的原本内容。而使用 eval 命令，经二次扫描，就可以看见了：

```
$ eval $MY_FILE
I love bash.
```

假设希望 a1=1，a2=2，…，a10=10，需要运行 10 条赋值命令，如果一直到 a100，那么需要运行 100 条重复性的命令。而运用 eval，则不需要运行那么多命令。在脚本 eval_1.sh 中，for 循环内的 eval 后面是赋值命令；脚本的最后三行，选取了三个变量，显示其值。

```
$ cat eval_1.sh
#!/bin/bash
for i in {1..100}
do
```

```
    eval a$i=$i
done
echo a3="$a3"
echo a16="$a16"
echo a78="$a78"
```

运行脚本：

```
$ eval_1.sh
a3=3
a16=16
a78=78
```

eval_1.sh 中的 eval 是必须的。把 eval_1.sh 中的 eval 去掉，得到脚本 eval_2.sh：

```
$ cat eval_2.sh
#!/bin/bash
for i in {1..100}
do
    a$i=$i
done
```

运行 eval_2.sh，出错：

```
$ eval_2.sh
./eval_2.sh: line 4: a1=1: command not found
./eval_2.sh: line 4: a2=2: command not found
......
./eval_2.sh: line 4: a100=100: command not found
```

eval 常用来取得最后一个参数。例如，脚本 eval_last_arg.sh 只包含一条命令：

```
$ cat eval_last_arg.sh
#!/bin/bash
eval echo \$$#
```

运行 eval_last_arg.sh Jan Feb 时，一共有两个位置参数，$#=2，eval 进行第一次扫描后，$# 被替换为 2，命令变为 echo $2，第二次扫描，即执行 echo $2，第二个位置参数 Feb 显示出来：

```
$ eval_last_arg.sh Jan Feb
Feb
```

有多个位置参数时：

```
$ eval_last_arg.sh Jan Feb Mar Apr May
May
```

Bash 是没有指针的，使用命令 eval 可以模拟指针。例如：

```
$ a=10
$ point_a=a                # point_a 是指向 a 的“指针”
$ eval echo \$$point_a
10
$ eval $point_a=60         # 第一次扫描，point_a 被替换为 a 得到 a=60，第二次则执行 a=60
$ echo $a
60                         # 是不是有些“指针的味道”
```

10.2　目录栈操作命令 pushd、popd 与 dirs

命令 cd 可用来改变工作目录，但它只能“记住”上一次的目录（运行命令 cd -或者 cd $OLDPWD 可以回到上一次的目录）。使用目录栈可以让更多的目录被“记住”。内置命令 pushd 用来将目录添加到目录栈，运行命令 pushd 之前，当前目录默认已经在栈里面。每添加一个目录后，当前路径默认变为刚刚添加的目录。因为 pushd 与 popd 命令默认会使当前路径发生变化，为了便于看清当前路径是如何随着 pushd 与 popd 命令的运行而变化的，所以下面的例子中的命令行提示符不再是一个单独的美元符$，$前面是当前的路径。

例如，一开始，当前路径为主目录~，添加/usr/bin 后，目录栈为/usr/bin ~：

```
user@ubuntu:~$ pushd /usr/bin
/usr/bin ~
```

再增加两个目录至目录栈，注意当前路径的变化：

```
user@ubuntu:/usr/bin$ pushd /tmp
/tmp /usr/bin ~
user@ubuntu:/tmp$ pushd /var/lib
/var/lib /tmp /usr/bin ~
user@ubuntu:/var/lib$
```

内置命令 dirs 用来显示目录栈。使用选项-c 时，目录栈将被清空（clear）。使用选项-p 时，每行只显示一个目录。使用选项-v 时，每行只显示一个有编号的目录，编号从 0 开始，编号小的在栈顶，编号大的在栈底。接着前面的例子：

```
user@ubuntu:/var/lib$ dirs
/var/lib /tmp /usr/bin ~
user@ubuntu:/var/lib$ dirs -p
/var/lib
/tmp
/usr/bin
~
user@ubuntu:/var/lib$ dirs -v
 0  /var/lib
 1  /tmp
 2  /usr/bin
 3  ~
```

内置变量 DIRSTACK（见表 4-3）记录了目录栈，它实际上是个数组，用 declare -p 命令可以查看它：

```
user@ubuntu:/var/lib$ declare -p DIRSTACK
declare -a DIRSTACK='([0]="/var/lib" [1]="/tmp" [2]="/usr/bin" [3]="/home/user")'
```

内置命令 popd 用来从目录栈中移除目录，默认从栈顶开始、每次移除一个目录。移除栈顶目录后，刚刚还处在第二位的目录变为栈顶，当前目录就变为新的处在栈顶的目录。接着前面的例子：

```
user@ubuntu:/var/lib$ popd          # 将栈顶/var/lib 弹出，/tmp 变为新的栈顶
/tmp /usr/bin ~
user@ubuntu:/tmp$ pwd
/tmp                                # 当前路径为处在栈顶的/tmp
user@ubuntu:/tmp$ popd              # 将栈顶/tmp 弹出，/usr/bin 变为新的栈顶
/usr/bin ~
user@ubuntu:/usr/bin$ popd
~
user@ubuntu:~$
```

当栈里面只剩下一个目录时，再继续运行 popd 命令时，会显示错误提示：

```
user@ubuntu:~$ popd
bash: popd: directory stack empty
```

命令 pushd 和 popd 运行时，当前路径默认会发生改变。这两条命令都有选项-n，使用该选项时，在添加或移除目录时，当前路径不改变。

本节讲的三条命令都可以带参数+N 和-N，N 为非负整数，从 0 开始取值。

命令 dirs ±N 不改变目录栈的内容。dirs +N 的作用是：显示从栈顶（或者说从左边）开始数的第 N 个目录；dirs -N 的作用是：显示从栈底（或者说从右边）开始数的第 N 个目录。

例如，目录栈有 4 个目录：

```
$ dirs
/opt /tmp /usr/bin /etc
$ dirs -v
 0  /opt
 1  /tmp
 2  /usr/bin
 3  /etc
$ dirs +0                           # 显示栈顶目录
/opt
$ dirs +1                           # 显示栈顶下面（右边）的第 1 个目录
/tmp
$ dirs -0                           # 显示栈底目录
/etc
```

```
$ dirs -1                        # 显示栈底上面（左边）的第 1 个目录
/usr/bin
$ dirs -3                        # 从 0 开始数，显示栈底上面（左边）的第 3 个目录
/opt
```

命令 popd 默认只从栈顶移除目录，而 popd ±N 可以有选择地移除目录。popd +N 的作用是：移除从栈顶（左边）开始数的第 N 个目录；popd -N 的作用是：移除从栈底（右边）开始数的第 N 个目录。

接着前面的例子，先显示目录栈，然后移除从栈顶开始（从 0 开始数）的第 1 个目录：

```
$ dirs -v
 0   /opt
 1   /tmp
 2   /usr/bin
 3   /etc
$ popd +1
/opt /usr/bin /etc               # 可见/tmp 已经被移除
```

移除栈底目录：

```
$ popd -0
/opt /usr/bin                    # 可见/etc 已经被移除
```

查看一下目录栈：

```
$ dirs -v
 0   /opt
 1   /usr/bin
```

命令 pushd ±N 并不向栈内添加目录，而是调整栈内的目录次序。把目录栈想象为栈顶与栈底首尾相接的“环”，pushd +N 的作用是：从栈顶（左边）开始的第 N 个目录变为栈顶；popd -N 的作用是：从栈底（右边）开始的第 N 个目录变为栈顶；刚刚还处在新的栈顶目录之上的（左边的）目录沿着“环”滑到栈尾重新“排队”，栈内目录次序的调整结束。命令 pushd ±N 的输出结果为“滑动”后的目录栈。下面举例说明。先显示目录栈：

```
$ dirs
/etc /opt /sbin /tmp
```

栈顶目录本来就在栈顶，pushd +0 不使栈发生变化：

```
$ pushd +0
/etc /opt /sbin /tmp
```

从 0 开始数，将第 1 个目录/opt 变为栈顶，原栈顶/etc“滑动”到栈底：

```
$ pushd +1
/opt /sbin /tmp /etc
```

现在目录栈为/opt /sbin /tmp /etc，将栈底目录/etc“提拔”为栈顶：

```
$ pushd -0
/etc /opt /sbin /tmp
```

现在目录栈为/etc /opt /sbin /tmp，将栈底上面的第 1 个目录/sbin 变为栈顶：

```
$ pushd -1
/sbin /tmp /etc /opt
```

现在目录栈为/sbin /tmp /etc /opt，将栈顶下面的第 2 个目录/etc 变为栈顶：

```
$ pushd +2
/etc /opt /sbin /tmp
```

利用 pushd 与 popd，可以从当前目录切换到另外一个目录，执行某条命令后，再返回当前目录，格式为：pushd dir1 && command1 && popd（其实这条命令可以由 cd dir1 && command1 && cd -来代替）。又如，要分别在两个目录下执行命令再返回到当前目录，命令格式为：pushd dir1 && command1 && pushd dir2 && command2 && popd && popd。

10.3　波浪号扩展

学习了命令 cd 和内置变量 HOME 之后，读者知道了波浪号~代表当前用户的主目录，波浪号后紧跟一个账户名代表该账户的主目录。其实波浪号的作用不仅是这两点。

~+表示当前目录，与内置变量 PWD 的值相同；~-表示上一次的工作目录，与内置变量 OLDPWD 的值相同。

~+和~-后面可以跟一个整数来表示目录栈中的目录。~+后面整数的意义与目录栈中目录的编号的意义一致，例如~+0 代表栈顶目录；~-后面整数的意义刚好相反，例如~-0 代表栈底目录。

例如，当前目录栈如下（其中~就是账户 user 的主目录/home/user）：

```
$ dirs -v
 0  /etc
 1  /usr/bin
 2  /tmp
 3  ~
$ echo ~+0                # ~+0 代表栈顶目录
/etc
$ echo ~+3                # ~+3 代表从 0 开始数，从栈顶向下数到 3 的目录
/home/user
$ echo ~+2
/tmp
$ echo ~-0                # ~-0 代表栈底目录
/home/user
$ echo ~-3                # ~-3 代表从 0 开始数，从栈底向上数到 3 的目录
/etc
```

~+后面跟一个整数来表示目录栈中的目录时，+可以省略。例如，~0 相当于~+0，表示栈顶目录，~1 相当于~+1，表示栈顶下面的目录。

10.4 （非）登录及（非）交互 shell

前面的内容中的命令都是在一个打开的 shell 窗口执行命令，这样的 shell 叫交互 shell。交互 shell 又分为交互登录 shell 和交互非登录 shell，也简称登录 shell 和非登录 shell。

登录 shell 是指：用户登录时，输入用户名和密码后登录的 shell；或者运行命令 bash --login 而启动的 shell。非登录 shell 不需要进行系统的认证，例如，通过图形界面打开一个 shell 终端，或者运行命令 bash 而启动的 shell。登录 shell 和非登录 shell 的主要区别是：启动 shell 时所读取并执行的文件不同。登录 shell 读取并执行/etc/profile 与~/.bash_profile、~/.bash_login 或者~/.profile（读取它们中的第一个存在的可读文件），还有~/.bash_logout（登出时）。非登录 shell 读取并执行~/.bashrc。

在第 3 章讲解搭建自己的环境时，为什么只介绍了.bashrc 而没有介绍其他文件呢？因为这几个文件只是名字不同，设置方法完全一样，而且文件.bash_profile 往往指向（或包含）.bashrc，在文件.bash_profile 或者文件.profile 里常常能看见：

```
if [ -f "$HOME/.bashrc" ]; then
    . "$HOME/.bashrc"                  # 或者 source "$HOME/.bashrc"
fi
```

与交互 shell 对应的是非交互 shell。执行脚本时，shell 就工作在非交互式模式下。在非交互模式下，bash 读取的启动文件由环境变量 BASH_ENV 来决定（见表 4-3）。

10.5 Bash shell 选项

用命令 bash 启动一个新的 shell 时，可以通过选项改变其行为。有两种类型的选项：短选项和长选项。短选项由一个连字符后面跟着一个字符构成，就是 1.6 节讲过的单字符选项，简称选项；长选项，即单词选项，由两个连字符后面跟着多个字符构成。同时使用时，长选项必须放在短选项之前，即格式为：

```
bash [长选项] [选项] [脚本]
```

Bash 的常用选项见表 10-1。

表 10-1　Bash 常用选项

选　　项	含　　义
-c string	读取 string 当作命令
-i	shell 在交互模式下运行
-s	从标准输入读取命令，并允许设置位置参数
-r	启动一个受限 shell
--	选项结束标志，不再处理后面的选项，把后面的内容当作文件名或者参数，即使它们以连字符开头

（续）

选　　项	含　　义
--help	显示帮助信息
--login	把 shell 作为一个登录 shell，同-l
--noediting	阻止用户在交互式 shell 中使用 Readline 库编辑命令行
--noprofile	阻止读取初始化文件/etc/profile、~/.bash_profile、~/.bash_login 和~/.profile
--norc	在交互式 shell，阻止读取~/.bashrc 初始化文件。如果 shell 以 sh 调用的话，这个选项默认是打开的
--posix	当 Bash shell 不符合 POSIX 标准时，就改变它使之符合
--rcfile file	对于交互 shell，使用指定的初始化文件 file 而不是~/.bashrc
--restricted	启动一个受限 shell
--verbose	打开后，显示其读入的每一行（无论是注释行、命令行还是空行），同-v
--version	显示版本信息

例如，要查看当前 Bash 的版本，可运行如下命令：

```
$ bash --version
GNU bash, version 4.2.24(1)-release (i686-pc-linux-gnu)
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

下面举例说明选项--noediting 的作用。输入一条命令，按回车键之前，可以编辑它。默认地，按〈Ctrl+A〉键回到行首，按〈Ctrl+E〉键移到行尾（end），按〈Ctrl+B〉键回退一个字符（back），按〈Ctrl+F〉键前进一个字符（forward），按〈Ctrl+←〉键（左方向键）回退一个单词，按〈Ctrl+→〉键（右方向键）前进一个单词。输入如下 echo 命令（不回车），再输入刚刚讲过的 6 种组合键试试：

```
$ echo "I study bash."
```

输入 bash --noediting 命令进入新的 shell，再输入那条 echo 命令（不回车），接着输入那几种组合键，会发现组合键都“不听使唤”了：

```
$ bash --noediting
$ echo "I study bash."
```

10.6　用命令 set 设置 shell

前面提到过内置命令 set，它可以显示当前 shell 的所有变量，还可以重新设置位置参数。set 的更重要的作用是设置当前 shell 的选项，3.9.3 节提到了其中的一个选项。

命令 set -o 的输出有两列，分别为选项名和相应的开关状态，off 表示状态为关闭，on 表示状态为打开。只关心某一个选项是开还是关的话，可用 grep 命令过滤它，如：

```
$ set -o | grep allexport
allexport       off
```

查询某个选项的开关状态，还有一种方法，格式为：

```
[ -o option_name]  或  test -o option_name
```

条件判断命令的退出状态为 0 时，表示选项是开着的，为非 0 时，表示选项是关闭的。例如，判断 allexport 是否打开的脚本可以这样写：

```
if [ -o allexport ]; then
    echo option allexport is opened
else
    echo option allexport is closed
fi
```

set -o 后面跟选项，即可打开该选项：

```
$ set -o allexport                          # 打开 allexport
$ set -o | grep allexport
allexport        on                         # 查看一下，allexport 打开了
```

set +o 后面跟选项，则关闭该选项：

```
$ set +o allexport                          # 关闭 allexport
$ set -o | grep allexport
allexport        off                        # 查看一下，allexport 关闭了
```

很多选项有快捷方式。如命令 set -a 的作用是打开 allexport，相当于 set -o allexport：

```
$ set -a
$ set -o | grep allexport
allexport        on                         # 查看一下，allexport 打开了
```

set +a 的作用是关闭 allexport，相当于 set +o allexport。set 命令可以控制的主要选项见表 10-2。

表 10-2　命令 set 的选项

选　项　名	快捷开关	含　　义
allexport	-a	从这个选项被设置开始就自动 export 新变量或修改过的变量，直至选项被复位
braceexpand	-B	打开大括号扩展，它是一个默认设置
emacs		使用 emacs 内置编辑器进行命令行编辑，是一个默认设置
errexit	-e	如果一个命令返回一个非 0 退出状态（失败），就退出程序
hashall	-h	在查找并执行命令时，记住它们的位置，是一个默认设置
histexpand	-H	执行历史替换时打开!和!!扩展，是一个默认设置
history		打开命令行历史功能，是一个默认设置
ignoreeof		禁止用 EOF（〈Ctrl+D〉键）键退出 shell，必须用 exit 命令退出
interactive-comments		对于交互式 shell，把#符后面的文本作为注释
keyword	-k	将关键字参数放到命令的环境中
monitor	-m	设置作业控制，在交互 shell 中默认是开启的

（续）

选 项 名	快捷开关	含 义
noclobber	-C	防止文件在重定向时被覆盖
noexec	-n	读命令，解释但不执行。用来检查脚本的语法。交互式运行时不开启
noglob	-f	禁用文件名和路径名扩展，即关闭通配符
notify	-b	后台作业完成时通知用户
nounset	-u	有未设置的变量时显示错误信息
onecmd	-t	在读取和执行命令后退出
physical	-P	设置后，运行命令（如 cd 或 pwd）时，禁止符号链接，用物理目录代替
posix		如果 shell 的默认操作不符合 POSIX 标准，就改变 shell 的行为，使之符合
privileged	-p	设置后，shell 不读取.profile 或 ENV 文件，且不从环境继承 shell 函数，将自动为 setuid 脚本开启特权
verbose	-v	为调试打开 verbose 模式，脚本运行时显示读入的每行命令，并原样打印
vi		使用 vi 成为内置编辑器进行命令行编辑
xtrace	-x	为调试打开 echo 模式，脚本运行时显示变量替换后的每行命令和参数

表中的选项较多，不一一讲解了，仅选择几个讲解一下。

解释一下选项 allexport 的作用。如果打开了 allexport，在当前的 shell 中定义了一个新变量或者修改一个老变量时，该变量会自动被导出。默认地，选项 allexport 是关闭的，需执行 export 命令才能导出变量。

举例讲一下选项 errexit 的作用。一个脚本当中某条命令失败了，后面的命令默认会照常执行。例如，脚本 errexit_1.sh 的第三行是一条错误命令：

```
$ cat errexit_1.sh
#!/bin/bash
echo "ABCDEFG"
aircondition
echo "UVWXYZ"
```

运行 errexit_1.sh，脚本的第 3 行出错，但是第 4 行命令照常执行：

```
$ errexit_1.sh
ABCDEFG
./errexit_1.sh: line 3: aircondition: command not found
UVWXYZ
```

在上面的脚本中增加 set -e 命令，就是打开选项 errexit，得到脚本 errexit_2.sh：

```
$ cat errexit_2.sh
#!/bin/bash
set -e
echo "ABCDEFG"
aircondition
echo "UVWXYZ"
```

执行 errexit_2.sh，运行到出错的地方，脚本就停止了，后面的命令不再执行：

```
$ errexit_2.sh
ABCDEFG
./errexit_2.sh: line 4: aircondition: command not found
```

选项 noglob 打开后则关闭路径名与文件名的通配符（no globbing）。命令 ls *默认是用来显示当前目录下所有的非隐藏文件和目录，打开选项 noglob 后，就不是这样了：

```
$ set -f                                # 或者 set -o noglob
$ ls *
ls: cannot access *: No such file or directory
```

打开选项 noglob 后，在 ls *命令里，*不再是通配符，命令 ls *试图列出名字就叫*的文件。这时可以产生一个名字就叫*的文件：

```
$ touch *
$ ls *
*
```

在一个 shell Terminal 里面，按〈Ctrl+D〉键，shell 就退出了。打开选项 ignoreeof 后，按〈Ctrl+D〉键，shell 窗口不会关闭，而得到"用 exit 离开 shell"的系统提示，这时可用 exit 命令退出 shell：

```
$ set -o ignoreeof
$ Use "exit" to leave the shell.        # 按〈Ctrl+D〉键得到的系统提示
$ exit                                  # 用 exit 命令退出 shell，用〈Ctrl+D〉键不能退出
```

选项 histexpand 默认是打开的，例如，!n 表示历史命令中的第 n 条命令，!-n 表示倒数第 n 条，!!表示最后一条（与!-1 相同），!string 表示以 string 开头的离现在最近的历史命令。

例如，先运行 HISTIGNORE=history，确保命令 history 本身不被记入命令历史中（见表 4-3），再运行 history -c 清空原有的历史命令。然后，依次执行 ls，pwd，date，uname 和 whoami 命令，现在命令历史如下：

```
$ history
    1  ls
    2  pwd
    3  date
    4  uname
    5  whoami
```

!!表示最后一条命令 whoami，!3 表示第 3 条命令 date，!un 表示以 un 开头的命令 uname：

```
$ !!
whoami
user
$ !3
date
```

```
Sat Jan 18 08:23:42 EST 2014
$ !un
uname
Linux
```

10.7 用命令 shopt 设置 shell

shopt 命令是 set 命令的扩展和补充，使用选项-p 来查看命令 shopt 选项的设置。

```
$ shopt -p
shopt -u autocd
shopt -u cdable_vars
shopt -u cdspell
shopt -u checkhash
shopt -u checkjobs
shopt -s checkwinsize
shopt -s cmdhist
shopt -u compat31
shopt -u compat32
shopt -u compat40
shopt -u compat41
shopt -u dirspell
shopt -u dotglob
shopt -u execfail
shopt -s expand_aliases
shopt -u extdebug
shopt -s extglob
shopt -s extquote
shopt -u failglob
shopt -s force_fignore
shopt -u globstar
shopt -u gnu_errfmt
shopt -s histappend
shopt -u histreedit
shopt -u histverify
shopt -u hostcomplete
shopt -u huponexit
shopt -s interactive_comments
shopt -u lastpipe
shopt -u lithist
shopt -u login_shell
shopt -u mailwarn
shopt -u no_empty_cmd_completion
shopt -u nocaseglob
shopt -u nocasematch
```

```
shopt -u nullglob
shopt -s progcomp
shopt -s promptvars
shopt -u restricted_shell
shopt -u shift_verbose
shopt -s sourcepath
shopt -u xpg_echo
```

上面命令的输出当中，-s 表示该选项是打开的（set），-u 表示该选项是关闭的（unset）。shopt 命令可以控制的主要选项见表 10-3。

表 10-3　命令 shopt 的选项

选　项	含　义
cdable_vars	如果给 cd 内置命令的参数不是一个目录，就假设它是一个变量名，变量的值是将要转换到的目录
cdspell	纠正 cd 命令中目录名的较小拼写错误，包括颠倒顺序的字符、遗漏的字符以及重复的字符。如果 Bash 知道如何纠正，就打印出正确的路径，命令将继续。只用于交互式 shell
checkhash	Bash 在试图执行一个命令前，先在哈希表中寻找，以确定命令是否存在。如果命令不存在，就执行正常路径搜索
checkwinsize	Bash 在每个命令后检查窗口大小，如果有必要，就更新内置变量 LINES 和 COLUMNS 的值
cmdhist	Bash 可以将一个多行命令的所有行保存在同一个历史项中。这使得多行命令的重新编辑更方便
dotglob	Bash 在文件名扩展的结果中包括以点（.）开头的文件名
execfail	如果一个非交互式 shell 不能执行指定给内置命令 exec 作为参数的文件，它不会退出。如果 exec 失败，一个交互式 shell 也不会退出
expand_aliases	别名被展开，对于交互 shell，默认为打开别名展开功能，对于非交互 shell，默认为关闭
extglob	打开扩展的模式匹配特征
histappend	当 shell 退出时，历史清单将添加到以 HISTFILE 变量的值命名的文件中，而不是覆盖文件
histreedit	如果 readline 正被使用，用户有机会重新编辑一个失败的历史命令替换
histverify	如果设置，且 readline 正被使用，历史替换的结果不会立即传递给 shell 解析器。而是将结果行装入 readline 编辑缓冲区中，允许进一步修改
hostcomplete	如果设置，且 readine 正被使用，当正在完成一个包含@的词时 Bash 将试图执行主机名补全（自动完成@字符后的主机名拼写）。默认为打开
huponexit	如果设置，当一个交互式登录 shell 退出时，Bash 将发送一个挂起信号 SIGHUP 给所有的作业
interactive_comments	在一个交互式 shell 中，允许以#开头的词以及同一行中其他的字符被忽略。默认为打开
lithist	如果打开，且 cmdhist 选项也打开，多行命令将用嵌入的换行符保存到历史中，而无需在可能的地方用分号来分隔
mailwarn	如果设置，且 Bash 用来检查邮件的文件自从上次检查后已经被访问，将显示消息“The mail in mailfile has been read”
nocaseglob	如果设置，当执行文件名扩展时，Bash 在不区分大小写的方式下匹配文件名
nullglob	如果设置，Bash 允许没有匹配任何文件的文件名模式扩展成一个空串，而不是它们本身
promptvars	如果设置，提示串在被扩展后再经历变量和参量扩展。默认为打开
restricted_shell	如果 shell 在受限模式下启动就设置这个选项。该值不能被改变。当执行启动文件时，不能复位该选项，允许启动文件发现 shell 是否是受限的
shift_verbose	设置后，如果内置命令 shift 移动的个数超过位置参数个数时，则 shift 命令打印出错信息
sourcepath	如果设置，运行 source file 时，source 命令在搜索路径（即内置环境变量 PATH 的值）中寻找作为参数的文件 file。默认为打开

命令 shopt -s <选项名>用来打开选项；命令 shopt -u <选项名>用于关闭选项。表 10-3 中的选项较多，下面只选择几个讲解。表格中的 extglob，在 8.2.4 节已经讲过。

先讲解 nocaseglob 的作用。在 Linux 下查看某个目录中文件时，文件名默认是区分字母大小写的，如：

```
$ touch FILE.txt file.txt
$ ls fi*
file.txt
```

打开选项 nocaseglob 之后，就不区分字母大小写了：

```
$ shopt -s nocaseglob
$ ls fi*
file.txt    FILE.txt
```

再看看 dotglob。命令 ls *默认不显示隐藏文件和隐藏目录，把选项 dotglob 打开后，再执行 ls *就可以查看到隐藏文件和隐藏目录（名字以点开头的文件和目录）了。

最后讲讲 expand_aliases。该选项在交互 shell 里默认是打开的，如果关闭它，已经定义的别名和之后定义的别名都不能被展开了（也可以这样说：如果关闭它，仍能继续定义别名，但所有的别名都不能被使用了）。例如，别名 ll 的定义如下：

```
$ alias ll
alias ll='ls -alF'
```

运行 ll 时，ll 默认可以被展开为命令 ls -alF 并执行。现在，关闭选项 expand_aliases，然后运行 ll 时，因为别名不能被展开，所以遇到命令未找到的提示：

```
$ shopt -u expand_aliases
$ ll
ll: command not found
```

10.8　终端行设置命令 stty

命令 stty 用来显示和修改终端行设置（set tty）。使用选项-a 或者--all 可以打印较详细的终端行设置：

```
$ stty -a
speed 38400 baud; rows 22; columns 103; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?; swtch = M-^?;
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclc ixany imaxbel iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt echoctl echoke
```

命令 stty size 可以显示终端的大小，即行数和列数：

```
$ stty size
```

```
22 103
```

stty 常用的设置项及其默认值如下。

intr：中断程序，默认为〈Ctrl+C〉键，按〈Ctrl+C〉键发送 INT 信号。

quit：退出程序，默认为〈Ctrl+\〉键，按〈Ctrl+\〉键发送 QUIT 信号。

erase：擦除最后一个输入字符，默认为〈Ctrl+?〉键，有的系统为〈Ctrl+H〉键。不过，多数用户更喜欢使用键盘上的退格键〈Backspace〉。

kill：删除整行命令，默认为〈Ctrl+U〉键。

eof：输入结束，文件结束，默认为〈Ctrl+D〉键。

stop：停止屏幕输出，默认为〈Ctrl+S〉键。

start：启动屏幕输出（如果停止），默认为〈Ctrl+Q〉键。

susp：挂起当前程序，默认为〈Ctrl+Z〉键，按〈Ctrl+Z〉键发送 TSTP 信号。

werase：删除最后一个单词，默认为〈Ctrl+W〉键。

例如，在 9.7 节，运行命令 tail -f latest_time.txt 可以看见时间在前进，屏幕在滚动。按下〈Ctrl+S〉键，屏幕就不动了（程序并未停止运行）。按〈Ctrl+Q〉键，屏幕继续滚动。

在命令行输入如下内容，不要按回车键，然后依次按〈Ctrl+W〉键、〈Ctrl+?〉键（或〈Ctrl+H〉键）和〈Ctrl+U〉键，可以体会到它们的作用：

```
$ cmd option par1 par2 par3
```

stty 的设置项的值是可以修改的，格式为：

```
stty  设置项  操作字符
```

例如，将中断程序的操作设置为〈Ctrl+P〉键，代替〈Ctrl+C〉键：

```
$ stty intr ^P
```

运行 cat 命令，输入 morning，看到 cat 命令的输出 morning 后，按〈Ctrl+P〉键退出：

```
$ cat
morning
morning
^P
```

stty 命令还有一些其他用法，如，命令 stty -echo 关闭回显，命令 stty echo 打开回显。在命令行，执行 stty -echo 之后的键盘输入看不见了，直到执行 stty echo 后，键盘输入才可以看得见。在脚本中需要输入密码时可以用 stty 命令，脚本片段如下所示：

```
echo "please input password"
stty -echo                    # 输入密码前关闭回显
read passwd
stty echo                     # 输入密码后打开回显
```

功能与命令 read -s passwd 的一致。

10.9 不在脚本和函数内使用别名

一行可以有多条命令，用分号隔开。但是，如果定义一个别名的命令和使用该别名的命令在同一行时，该别名是不生效的。例如：

```
$ alias mo='echo Good morning'; mo
mo: command not found              #  别名 mo 还未生效
```

因为定义别名 mo 的命令后面是分号，不是回车符，在那一刻别名 mo 还不生效。上面命令行的最后面才是回车符，遇到回车符后，别名才生效。

现在别名 mo 已经生效了，这时，再运行别名 mo 就可以了：

```
$ mo                               #  运行别名 mo
Good morning
```

上面的例子说明，不能让别名的定义和使用处在同一行。有关别名，还有更需要注意的地方。

在脚本里，别名定义默认是不起作用的。看脚本 alias_1.sh，定义了一个别名，然后使用该别名：

```
$ cat alias_1.sh
#!/bin/bash
alias aftn='echo Good afternoon'
aftn
```

运行脚本，发现运行到别名的使用处（最后一行）出错：

```
$ alias_1.sh
./alias_1.sh: line 3: aftn: command not found
```

因为在非交互 shell 里，选项 expand_aliases 默认是关闭的，见表 10-3。在脚本 alias_2.sh 里把该选项打开：

```
$ cat alias_2.sh
#!/bin/bash
shopt -s expand_aliases
alias aftn='echo Good afternoon'
aftn
```

运行脚本，别名就起作用了：

```
$ alias_2.sh
Good afternoon
```

如果一个别名定义在函数内，在该函数被调用前，别名是不生效的。看脚本 alias_in_fun_1.sh，先在函数内定义了别名，函数外使用了该别名：

```
$ cat alias_in_fun_1.sh
#!/bin/bash
shopt -s expand_aliases
function alias_fun()
{
	alias wkd='echo Have a good weekend'
}
wkd
```

运行脚本，最末一行的别名不起作用：

```
$ alias_in_fun_1.sh
./alias_in_fun_1.sh: line 7: wkd: command not found
```

修改脚本 alias_in_fun_1.sh，在使用别名之前增加函数调用，得到脚本 alias_in_fun_2.sh：

```
$ cat alias_in_fun_2.sh
#!/bin/bash
shopt -s expand_aliases
function alias_fun()
{
	alias wkd='echo Have a good weekend'
}
alias_fun			# 调用函数
wkd				# 这时，别名生效了
```

运行脚本，最末一行的别名起作用了：

```
$ alias_in_fun_2.sh
Have a good weekend
```

别名不能导出，export 命令不能用在别名上。父 shell 里面定义好的别名，到了子 shell 是不起作用的。有个办法，将希望在子 shell 里能够使用的别名的定义放在一个文件里，例如：

```
$ cat alias_list.txt
alias mo='echo Good morning'
```

然后在子 shell 里（如脚本里）运行 source <别名列表文件>，就可以使用别名了：

```
$ cat alias_in_sub_shell.sh
#!/bin/bash
shopt -s expand_aliases
source alias_list.txt
mo
```

运行脚本，最末一行的别名起作用：

```
$ alias_in_sub_shell.sh
```

```
Good morning
```

用户自己的别名一般是定义在~/.bashrc 文件里面。使用别名几乎都是为了缩短命令，容易记忆，加快执行。在脚本里面使用别名是不必要的，用一条短命令代替长命令的操作，用函数也可以实现。

综上所述，建议不要在脚本和函数内使用别名。

10.10　Bash 调试

Bash 有一定的调试功能。命令 set 和 shopt 有几个选项具有调试作用。

Bash 默认可以引用一个未定义的变量。例如，显示一个未定义的 y 时，得到的是空行，不算是错误：

```
$ echo $y
                                        # 显示空行
```

表 10-2 中的选项 nounset 打开后，Bash 不允许引用未定义的变量，否则会提示出错。运行 set -u 或 set -o nounset 打开这个选项后，再试着显示未定义的变量 y：

```
$ set -u
$ echo $y
bash: y: unbound variable               # 系统提示出错
```

如果一个 Bash 脚本能像其他高级语言那样，必须先定义变量才能引用变量，那么这个脚本会相对严谨一些，为脚本的调试打下一个良好的基础。

一个脚本内命令 shift 的左移参数的次数超过参数个数时，默认脚本运行不会出错。当表 10-3 中的选项 shift_verbose 打开，并且 shift 的移动参数次数超过参数个数时，运行脚本会遇到出现错误的提示。所以，在使用 shift 命令较多的脚本里，打开选项 shift_verbose，会使脚本编写得更严谨一些。

一个脚本通常是由多条命令组成的，表 10-2 中的选项 xtrace 打开后，脚本中的每一条命令及其参数都看得到，如有循环，每次循环的情况也都看得一清二楚。

脚本 debug_1.sh 中的“while :”循环本身是无穷循环，因为空命令“:”总是成功。每次循环，a 的值增长 1，当 a>=5 时，执行命令 break，循环退出。

```
$ cat debug_1.sh
#!/bin/bash
a=1
while :
do
    [ $a -ge 5 ] && break
    let a=$a+1
done
```

执行脚本 debug_1.sh 将没有任何输出，只能脑子里想象着 5 次 while 循环的情况。如果

在脚本的第二行增加命令 set -x 或者 set -o xtrace，脚本是怎么运行的就看得一清二楚。

```
$ cat debug_2.sh
#!/bin/bash
set -x                              #  在第二行增加命令 set -x
a=1
while :
do
    [ $a -ge 5 ] && break
    let a=$a+1
done
```

运行 debug_2.sh，每轮循环中的空命令“:”、中括号判断命令、赋值命令，和最后退出时的 break 命令都显示出来：

```
$ debug_2.sh
+ a=1
+ :
+ '[' 1 -ge 5 ']'
+ let a=1+1
+ :
+ '[' 2 -ge 5 ']'
+ let a=2+1
+ :
+ '[' 3 -ge 5 ']'
+ let a=3+1
+ :
+ '[' 4 -ge 5 ']'
+ let a=4+1
+ :
+ '[' 5 -ge 5 ']'
+ break
```

选项 verbose 打开后，一个脚本中的每条命令在执行之前会先原样显示出来。下面的脚本中有命令 set -xv 将选项 xtrace 和 verbose 同时打开：

```
$ cat debug_xv.sh
#!/bin/bash
set -xv
y=$1                        #  脚本的第一个参数存入变量 y
echo $y
```

运行该脚本，可以理解选项 verbose 的作用：

```
$ debug_xv.sh 85
y=$1                        #  脚本中的命令原样显示
+ y=85                      #  参数 85 赋给 y
```

```
echo $y                        # 脚本中的命令原样显示
+ echo 85
85
```

参数换为 Sunday，再运行一次：

```
$ debug_xv.sh Sunday
y=$1                           # 脚本中的命令原样显示
+ y=Sunday
echo $y                        # 脚本中的命令原样显示
+ echo Sunday
Sunday
```

当运行 set -x 之后，脚本一运行会看见很多+，这是为什么呢？它是表 4-3 中的 PS4 的值，PS4 的默认值为+，查看一下：

```
$ echo $PS4
+
```

像 PS1、PS2 和 PS3 一样，PS4 的值也是可以设置的。

脚本 debug_PS4.sh 的作用是：读入用户输入的名字，然后将当前目录下的名字以.txt 结尾的文件移到临时文件夹里面。

```
$ cat debug_PS4.sh
#!/bin/bash
set -x
read -p "please input user name:" user
for i in *.txt
do
      mkdir -p /tmp/$user
      mv $i /tmp/$user
done
```

查看一下当前目录下的.txt 文件，假设有如下几个文件：

```
$ ls *.txt
a1.txt    a2.txt    a3.txt
```

重新设置内置变量 PS4，其中 LINENO 也是内置变量，见表 4-3，它是脚本中命令所在的行号：

```
$ export PS4='+[$LINENO]'
```

运行脚本 debug_PS4.sh，键盘输入名字 Mike：

```
$ debug_PS4.sh
+[3]read -p 'please input user name:' user
please input user name:Mike                # 键盘输入 Mike
```

```
+[4]for i in '*.txt'
+[6]mkdir -p /tmp/Mike
+[7]mv a1.txt /tmp/Mike
+[4]for i in '*.txt'
+[6]mkdir -p /tmp/Mike
+[7]mv a2.txt /tmp/Mike
+[4]for i in '*.txt'
+[6]mkdir -p /tmp/Mike
+[7]mv a3.txt /tmp/Mike
```

从上面的输入可以发现脚本的一个小缺陷，就是命令 mkdir -p 应该放到 for 循环的上面，否则该命令会运行多次（重复创建同一个目录），这没必要，运行一次就行。

如果将脚本 debug_PS4.sh 中的命令 set -x 删除，运行命令 bash -x debug_PS4.sh，效果是一样的。即，运行命令 bash -x <脚本>，相当于在脚本中加了 set -x。

还有一种“省略”set -x 的方法，把脚本 debug_PS4.sh 的第一行改为#!/bin/bash -x，第二行的命令 set -x 就可以去除了。

有的时候，一个脚本特别长，而只需要对它的一部分进行调试，那么在需要调试的部分的前面加上命令 set -x（打开选项 xtrace），后面加上命令 set +x（关闭选项 xtrace）即可：

```
……
set -x                # 打开调试开关
    [脚本需要调试的部分]
set +x                # 关闭调试开关
……
```

除了 LINENO 之外，还有 BASH_SOURCE、BASH_SUBSHELL 和 FUNCNAME 等内置变量也可以用于脚本的调试。读者还可以安装 bashdb——专用于 Bash 脚本的调试工具。它有很强大的功能，如，设置断点、单步跟踪执行、变量观察等。

关于 Bash 调试的内容很丰富，并非只有本节所介绍的这些，这里再提几句。可以利用命令 trap 来调试，还可以在脚本中多增加打印命令，利用输出重定向来获取 log（脚本日志），通过看 log 来分析脚本的运行情况。还可以在脚本中增加“调试块”，当变量 DEBUG 声明之后，脚本能够打印调试信息。

```
if [ $DEBUG = YES ]; then
     调试信息…
fi
```

10.11　并行命令 parallel

在工作中有时需要处理非常大的数据文件（几十 GB，几百 GB 甚至更大），但处理数据的常用命令 grep，wc，awk 和 sed 等都是单线程的，只能使用一个 CPU 内核，这些命令在处理大数据时会比较慢。服务器往往有多个 CPU，要想让 Linux 命令使用所有的 CPU 内核，只要用 GNU parallel 命令，负载就会分配到各个 CPU 上，使得命令执行速度加快。

查看 CPU 相关信息的命令是 mpstat，例如，在某台 Linux 服务器上运行该命令：

```
$ mpstat
Linux 3.2.0-27-generic (ubuntuland719)    12/01/2013    _x86_64_    (24 CPU)

02:20:01 AM  CPU   %usr   %nice   %sys %iowait   %irq   %soft  %steal  %guest   %idle
02:20:01 AM  all   3.98    0.00   0.30    0.12   0.00    0.00    0.00    0.00   95.60
```

还可以通过查看文件/proc/cpuinfo 获得 CPU 信息，例如，查看逻辑 CPU 的个数：

```
$ cat /proc/cpuinfo  | grep processor | wc -l
24
```

GNU parallel 命令的下载与安装很简单，以下载 20131122 版本为例，步骤如下：

1）下载，运行命令 wget http://ftpmirror.gnu.org/parallel/parallel-20131122.tar.bz2；或者在 Windows 中下载，网址为 http://ftp.gnu.org/gnu/parallel/，然后复制到 Linux 里面；也可直接在 Linux 中用 Web 浏览器（如 Firefox）下载，下载后默认存放在~/Downloads 目录里。

2）解压缩，运行命令

```
bzip2 -dc parallel-20131122.tar.bz2 | tar xvf -
```

3）运行

```
cd parallel-20131122
```

4）安装，如果是 root 账户，运行命令./configure && make && make install；如果是普通账户，运行命令./configure --prefix=$HOME && make && make install，然后确保~/bin 目录在搜索路径里（即，运行 PATH="$PATH:~/bin"）。

下面举例说明关于 parallel 的应用。在某服务器上有一个经过压缩的大文件，看一下它的大小：

```
$ du -sh wk45.bin
11G     wk45.bin
```

将它压缩为 bz2 文件的命令是 cat wk45.bin | bzip2 --best > wk45.bz2，运行时间一定不短。为了计时，运行如下命令：

```
$ date > start.txt;cat wk45.bin | bzip2 --best > wk45.bz2; date > end.txt
```

经过“漫长”的等待，命令结束，查看用时，用了 32min43s 的时间：

```
$ cat start.txt end.txt
Sat Nov 30 22:54:15 CST 2013
Sat Nov 30 23:26:58 CST 2013
```

使用 time 命令，即运行 time cat wk45.bin | bzip2 --best > wk45.bz2，像 9.5 节中的例子那样，也可以计时。

把压缩命令改为：cat wk45.bin | parallel --pipe --recend '' -k bzip2 --best > wk45.bz2，计时命令不变，再查看用时，只用了 4min6s 的时间：

```
$ cat start.txt end.txt
Sat Nov 30 23:42:10 CST 2013
Sat Nov 30 23:46:16 CST 2013
```

确实快了很多。通过时间比较，基本说明了 parallel 的优点，但很难精确地说明快了多少倍，因为服务器上账户多、任务多，服务器不会单单被某一个账户专用。

下面看一个有关 wc 命令的例子。先要找到一个很多行的文件，也可以自己产生一个多行文件。例如，产生一个一百万行的文件 num1000000：

```
$ for i in {1..1000000}; do echo $i >> num1000000; done
```

在某台服务器上对一个一亿行的文件进行行数、单词数和字节数统计，并计时：

```
$ date > start.txt; cat num100000000 | wc; date > end.txt
100000000 100000000 888888898
$ cat end.txt start.txt
Sun Dec   1 00:06:09 CST 2013
Sun Dec   1 00:05:50 CST 2013
```

可见 wc 命令用时为 19s。把命令改为：

```
cat num100000000 | parallel --pipe wc | awk '{l+=$1;w+=$2;c+=$3} END {print l, w, c}'
```

计时命令不变，再查看用时，只用了 6s 的时间：

```
$ cat end.txt start.txt
Sun Dec   1 00:09:51 CST 2013
Sun Dec   1 00:09:45 CST 2013
```

grep 命令过滤某模式的命令为：

```
grep pattern bigfile.txt
```

使用 parallel 时相应的命令为：

```
cat bigfile.txt | parallel --pipe grep 'pattern'
```

sed 对某文件进行模式查找替换的命令为：

```
sed s/old/new/g bigfile.txt
```

使用 parallel 时相应的命令为：

```
cat bigfile.txt | parallel --pipe sed s/old/new/g
```

总之，在对大文件进行操作时感觉慢的话，可试着使用 parallel 命令。若想得到有关它的较详细讲解，可访问https://www.gnu.org/software/parallel/parallel_tutorial.html。

10.12 模拟旋转型进度指示

字符界面下的进度指示有打点方式，还有旋转指示。旋转指示就是四个字符|、/、-和\依次较快速地出现在同一位置，不断循环，给人以旋转的感觉。看脚本 progress.sh，它由一个

函数 progress 和主程序组成。在主程序里，有一个 while 循环，i 从 1 循环到 100，i 为函数 progress 的第一个参数。

```
$ cat progress.sh
#!/bin/bash
progress()
{
  _per=`expr $1 % 4`
  case $_per in
    0) _char="|" ;;
    1) _char="/" ;;
    2) _char="-" ;;
    3) _char="\\" ;;                # 反斜杠是特殊字符，显示一个反斜杠用\\
  esac
  printf "\r $_char $1 %%"          # \r 的作用是使光标回到行首

  if [ $1 -eq $2 ];then
    printf "\n"
  fi
}

#Main script                        # 主程序
i=1
while (( $i <= 100 ))
do
  sleep 0.1
  progress $i 100                   # 调用函数 progress
  i='expr $i + 1'
done
```

函数 progress 中第一条命令的作用是：i 被 4 除，取余数，将余数赋给变量_per。当 i=1 时，_per=i/4%4=1，_char=/；当 i=2 时，_per=i%4=2，_char=-；当 i=3 时，_per=i%4=3，_char=\；当 i=4 时，_per=i%4=0，case 命令中的_char=|……以此类推。

函数 progress 的第一个 printf 中的\r 的作用是让光标回到行首，保证四个字符|、/、-和\出现在同一位置，给人以旋转感。如果去掉\r，光标就不能回到行首，运行结果变成了这样（去掉\r 之后的脚本为 progress_no_return.sh）：

```
$ progress_no_return.sh      # 去掉 printf 中的\r，显示结果如下
 / 1 % - 2 % \ 3 % | 4 % / 5 % - 6 % \ 7 % | 8 % / 9 % - 10 % \ 11 % | 12 % / 13 % - 14 % \ 15 %
 | 16 % / 17 % - 18 % \ 19 % | 20 % / 21 % - 22 % \ 23 % | 24 % / 25 % - 26 % \ 27 % | 28 % / 29 % -
 30 % \ 31 % | 32 % / 33 % - 34 % \ 35 % | 36 % / 37 % - 38 % \ 39 % | 40 % / 41 % - 42 % \ 43 % |
 44 % / 45 % - 46 % \ 47 % | 48 % / 49 % - 50 % \ 51 % | 52 % / 53 % - 54 % \ 55 % | 56 % / 57 % -
 58 % \ 59 % | 60 % / 61 % - 62 % \ 63 % | 64 % / 65 % - 66 % \ 67 % | 68 % / 69 % - 70 % \ 71 % |
 72 % / 73 % - 74 % \ 75 % | 76 % / 77 % - 78 % \ 79 % | 80 % / 81 % - 82 % \ 83 % | 84 % / 85 % -
 86 % \ 87 % | 88 % / 89 % - 90 % \ 91 % | 92 % / 93 % - 94 % \ 95 % | 96 % / 97 % - 98 % \ 99 % |
 100 %
```

这个结果也不错，可以看清字符随着百分比的增长是如何变化的。

函数 progress 的第二个参数的值是固定值 100。函数 progress 的第二个 printf 命令的作用是：当第一个参数与第二个参数相等时，也就是当第一个参数为 100 时（即从 1 循环到最后一个值时），打印换行符\n。

主程序中的命令 sleep 0.1 不是必须的，去掉它的话，脚本运行时，百分比变化很快，一下子就从 1%变为 100%。有了命令 sleep 0.1，脚本运行时，可以看见百分比变化过程。脚本最终运行结果为：

```
$ progress.sh
 | 100 %              # 可以看见 1%到 100%的变化过程，最后显示 100%
```

10.13　删除文件空行

下面编写一个用来删除空行的脚本 del_blank.sh，它可以处理一个或多个文件。当一个或者一个以上包含空行的文本文件为脚本的参数时，运行后，文本文件中的空行被删除，其他内容不受影响。

脚本 del_blank.sh 的前面为函数 Usage 的定义，后面是主程序。主程序中，如果发现脚本没有参数，即$#等于 0 时，调用函数 Usage，显示帮助信息，脚本结束。

如果脚本参数个数$#大于 0，则由 while 循环和左移参数的 shift 命令配合对参数逐个处理。如果参数是一个存在的普通文件，用 sed 命令删除其空行，否则，显示提示信息（文件不存在，不是普通文件，是目录等）。第一个参数等于-help、-h 或--help 时，调用函数 Usage，脚本结束。

```
$ cat del_blank.sh
#!/bin/bash
Usage()
{
   echo "Use this script to remove all blank lines from a text file(s)"
   echo "Usage : $(basename $0) file [file...]"
   exit 0
}

#Main script
if [ $# -eq 0 ]; then
   Usage
fi

TMP=/tmp/del_blank_$$                    # 临时文件，过渡用
while [ $# -gt 0 ]
do
   case $1 in
      -help|-h|--help)
         Usage
```

```
            ;;
        *)FILE=$1
            if [ -f $FILE ]; then
                sed '/^$/d' $FILE > $TMP          # 删除空行，重定向到临时文件
                mv -f $TMP $FILE
            else
                echo "cannot find this file $FILE, make sure it exists"
                echo "make sure it is not a directory, it should be a regular text file"
            fi
            shift
            ;;
    esac
done
```

查看脚本的帮助：

```
$ del_blank.sh -h
Use this script to remove all blank lines from a text file(s)
Usage : del_blank.sh file [file...]
```

给脚本一个目录，而不是文本文件：

```
$ del_blank.sh /tmp
cannot find this file /tmp, make sure it exists
make sure it is not a directory, it should be a regular text file
```

文件 test1.txt 有一个空行：

```
$ cat test1.txt
aaa
bbb

dddd
```

将 test1.txt 作为脚本 del_blank.sh 的参数，运行如下命令：

```
$ del_blank.sh test1.txt
```

文件 test1.txt 的空行消失了：

```
$ cat test1.txt
aaa
bbb
dddd
```

10.14 完善 while-shift 循环

前面章节的某些脚本只是用于讲解知识点，并不“完美”。以 6.6 节的 while_shift_2.sh

为例，当它后面跟奇数个参数时，不能正常运行，试着运行 while_shift_2.sh 10 20 30 40 50 就知道了。因为一开始有 5 个参数，$#=5，左移 2 个，$#=3，左移 2 个，$#=1，若再左移 2 个，这时剩余 1 个参数，却要移动 2 个，所以脚本运行异常。

如何预防和处理左移个数超出范围的问题呢？见表 10-3，如果在脚本的第二行增加命令 shopt -s shift_verbose，当左移个数超过剩余的参数个数时，用户会得到提示。当左移个数超出范围时，shift 命令的退出状态是非 0，可以对 shift 的退出状态进行判断并采取相应措施。修改后的脚本为 while_shift_2_improve.sh：

```
$ cat while_shift_2_improve.sh
#!/bin/bash
shopt -s shift_verbose
while [ $# -gt 0 ]
do
    echo $1; shift 2
    [[ $? != 0 ]] && exit
done
```

脚本参数个数为 5（奇数）时，运行结果如下，系统给出了“移动个数超出范围”提示：

```
$ while_shift_2_improve.sh 10 20 30 40 50
10
30
50
./while_shift_2_improve.sh: line 5: shift: 2: shift count out of range
```

附录　Bash 安全漏洞 Shellshock

2014 年 9 月下旬，网上疯传着 Bash 被发现了一个严重漏洞的消息。该漏洞被命名为 Shellshock（弹震症），编号是 CVE-2014-6271（CVE，Common Vulnerabilities & Exposures，公共漏洞披露），是由 Akamai 公司的 Stephane Chazelas 发现的。Shellshock 影响 Bash 的 1.14 到 4.3 版本，它“潜伏”了至少 20 年。下面来认识一下 Shellshock。

1．漏洞的检查与 env 命令

运行如下检测命令，如果输出结果为 vulnerable，则说明正在使用的 Bash 有漏洞：

```
$ env x='() { :;}; echo vulnerable' bash -c 'true'
vulnerable
```

由表 10-1 可知，bash -c 后面的字符串会被视作命令而执行。要理解该漏洞，需要学习一下命令 env。env 并不仅仅用于显示环境变量，它主要用于在构建的环境中执行命令，其基本格式为：

```
env [选项] [变量定义] [要运行的命令]
```

例如，给环境变量 EMAIL 赋值，用于向邮件地址（EMAIL 的值）发送日志：

```
$ export EMAIL='user@abc.com'            #  定义环境变量 EMAIL
$ env | grep -i EMAIL                    #  查看 EMAIL 的值，用 echo $EMAIL 也可以
EMAIL=user@abc.com
```

假设有一个特殊的日志，需发送到一个特殊的邮件地址，并且只发送这一次，可以使用如下命令：

```
$ env EMAIL='another_name@xyz.com' bash -c 'echo $EMAIL'
another_name@xyz.com
```

将上面命令中的 echo $EMAIL 替换为发送特殊日志的脚本即可达到目的。处理完特殊日志后，再查看当前环境中 EMAIL 的值，发现它没有变，不受上面命令的影响：

```
$ env | grep -i EMAIL                    #  查看 EMAIL 的值，用 echo $EMAIL 也可以
EMAIL=user@abc.com
```

讲到这里，读者应该理解命令 env 的功能了：它用于在构建的环境中执行命令。

2．Bash 对环境变量解析的失误

下面的命令将变量 x 赋值为一个函数的定义。要使函数生效，需用 eval 命令“扫描” x，然后调用函数 fun_b，那么函数中的 echo 命令可以运行：

```
$ env x='fun_b() { echo I am in fun_b;}' bash -c 'eval $x; fun_b'
I am in fun_b
```

下面的命令将变量 x 赋值为一个函数的定义连同命令 echo after fun_b（用分号隔开），用 eval 命令“扫描”x 后函数生效（扫描的同时，命令 echo after fun_b 被执行，所以先显示 after fun_b），然后调用函数 fun_b：

```
$ env x='fun_b() { echo I am in fun_b;}; echo after fun_b' bash -c 'eval $x; fun_b'
after fun_b
I am in fun_b
```

为了理解漏洞，还需看一看被导出的函数在环境中的“样子”：

```
$ fun_a() { echo in fun_a;}          # 定义函数 fun_a
$ export -f fun_a                    # 导出函数
$ env                                # 用 env 命令查看
fun_a=() {    echo in fun_a          # 输出很长，这里只保留相关的内容
}                                    # 注：这里是在有漏洞的 Bash 下的执行结果
```

从表面看，fun_a 是个普通的、只不过是以“() { ”等特殊字符开头的环境变量。那么，定义一个这种特殊字符开头的环境变量，是不是就会被 Bash 解释为函数呢？试一试：

```
$ env fun_x='() { echo in fun_x;}' bash -c 'fun_x'
in fun_x                             # 注：这里是在有漏洞的 Bash 下的执行结果
```

看来用特殊字符开头的字符串来定义函数的方法是可行的，而且省去了 eval 命令的扫描。在特殊字符开头的字符串的后半部加一条其他命令（用分号隔开）：

```
$ env fun_x='() { echo in fun_x;}; echo after fun_x' bash -c 'fun_x'
after fun_x                          # 注：这里是在有漏洞的 Bash 下的执行结果
in fun_x
```

命令 echo after fun_x 被执行了吗？根据 env 命令的格式，上面命令中的 fun_x='......'只是变量赋值，引号之间是字符串，echo after fun_x 是字符串的后半部，这里并没有用 eval 扫描变量，但是变量赋值的后半部竟然被解释为命令并被执行了。漏洞出现了！如果将上面命令中的 echo in fun_x 换为 null 命令，即冒号，再将 after fun_x 换为 vulnerable、将 fun_x 换为 true，就得到了一开始提到的检测漏洞的命令。

为什么说这是个漏洞呢？假设某系统的普通用户以环境变量的形式通过某种途径向系统提供自己的 email 地址，用于接收公共信息。普通用户没有权限控制或进入系统，用户自己负责 email 地址的正确性，系统不检查。如果有居心不良的人提供了恶意的 email 地址，就可能使系统遭到破坏。下面模拟一下，运行如下命令，使系统得到一个奇特的 email 地址：

```
$ export EMAIL='() { :;}; echo you are stupid'
```

系统中有一个与 email 地址不相关的脚本，内容如下：

```
$ cat m1.sh
#!/bin/bash
echo only_test
```

执行这个不相关的脚本，或者运行 bash 进入子 shell，都会看到 you are stupid 这句话：

```
$ m1.sh                              # 或者运行./m1.sh
```

```
you are stupid                         # 注：这里是在有漏洞的 Bash 下的执行结果
only_test
$ bash
you are stupid
```

因为系统在启动一个子 shell 时，会将父 shell 的环境变量带入，当 Bash 对这个不一般的环境变量解析失误时，变量 EMAIL 中的 echo you are stupid 就会执行。如果将 echo you are stupid 替换为破坏性的命令，如 rm -fr *，那么，没有权限控制系统的人也能轻易地摧毁系统。

通过前面的分析可知，只要是接受通过某种方法传递过来的环境变量的 Bash 程序都有可能受此影响，看来 Shellshock 的确是一个严重的漏洞。

3．修补漏洞

在 Shellshock 被发现后的短时间内，各大公司迅速修复了漏洞。主流 Linux 发行版，如 Red Hat、Fedora、Ubuntu 和 Debian 都发布了补丁。以 Ubuntu 为例，运行 sudo apt-get install bash（或 sudo apt-get install --only-upgrade bash）升级 Bash 之后，再运行如下命令，就得到了无漏洞的正确结果：

```
$ env x='() { :;}; echo vulnerable' bash -c 'true'
$ env x='() { :;}; echo vulnerable' bash -c 'echo hello'
hello
$ env fun_x='() { echo in fun_x;}' bash -c 'fun_x'
bash: fun_x: command not found
$ env x='() { :;}; echo vulnerable' bash -c 'echo $x'
() { :;}; echo vulnerable
```

解决漏洞的首选方法就是尽快升级 Bash。其次，应检查所有的脚本，监测环境变量，如发现变量值是以“() { ”开头就要做特殊处理，但这种方法工作量大，难免有疏漏。

网络上有更广泛、更深入的针对 Shellshock 的分析，感兴趣的读者可以参考：

http://www.solidot.org/story?sid=41266

http://www.solidot.org/story?sid=41285

http://devops.com/blogs/shellshock-vulnerability-impact-analysis-protection/

http://www.ubuntu.com/usn/USN-2362-1/

https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2014-6271

https://www.debian.org/security/2014/dsa-3032

https://securityblog.redhat.com/2014/09/24/bash-specially-crafted-environment-variables-code-injection-attack/

https://raw.githubusercontent.com/citypw/DNFWAH/master/4/d4_0x07_DNFWAH_shellshock_bash_story_cve-2014-6271.txt

http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271

http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271

http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7169

参 考 文 献

[1] 唐华．Linux 操作系统高级教程[M]．北京：电子工业出版社，2008．

[2] Stephen G Kochan，Patrick Wood．Unix Shell 编程[M]．袁科萍，译．北京：中国铁道出版社，2004.

[3] Arnold Robbins，Nelson Beebe．Shell 脚本学习指南[M]．北京：机械工业出版社，2009．

[4] Ron Peters．Shell 脚本专家指南[M]．李晓群，译．北京：人民邮电出版社，2010．

[5] Dale Dougherty，Arnold Robbins．sed 与 awk[M]．张旭东，译．北京：机械工业出版社，2003．

[6] Ellie Quigley．UNIX Shell 范例精解[M]．李化，译．北京：清华大学出版社，2007．

[7] Cameron Newham．Learning the Bash Shell[M]．3rd ed．Sebastopol: O'Reilly Media，2005．

[8] Machtelt Garrels．Bash Guide for Beginners[OL]．http://tille.garrels.be/training/bash/.

[9] Free Software Foundation．Bash Manual[OL]．http://www.gnu.org/software/bash/manual/.

[10] Aris．Use multiple CPU Cores with your Linux commands[OL]．http://www.rankfocus.com/.

[11] Free Software Foundation．GNU sed user's manual[OL]．http://www.gnu.org/software/sed/manual/.

在线互动交流平台
官方微博：http://weibo.com/cmpjsj
官方博客：http://blog.sina.com.cn/cmpbookjsj
读者信箱：cmp_itbook@163.com

实用Linux Shell编程

内容提要

本书系统地介绍了在 Linux 系统中广泛使用的 Bash 脚本语言。全书内容的安排由浅入深，体系合理。先讲解脚本的概念和学习环境的搭建，接下来介绍 Linux 的常用命令，然后根据概念之间的依赖关系，讲解 Bash 环境设置、变量与数组、条件流程控制、循环、函数、正则表达式、文本处理、进程与作业、高级话题等。本书是一本不可多得的 shell 编程原创读物。



书中代码可在 http://www.cmpbook.com 下载。

地址：北京市百万庄大街22号
邮政编码：100037
电话服务
社服务中心：010-88361066
销售一部：010-68326294
销售二部：010-88379649
读者购书热线：010-88379203
网络服务
教材网：http://www.cmpedu.com
机工官网：http://www.cmpbook.com
机工官博：http://weibo.com/cmp1952
封面无防伪标均为盗版

上架建议 计算机/程序设计

ISBN 978-7-111-48202-4

策划编辑◎车　忱

封面设计◎



扫一扫加入微信公众平台
更多、更快获取最新图书资讯



ISBN 978-7-111-48202-4



定价：52.00元